UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

TRABAJO FIN DE GRADO

Un editor gráfico de modelos en Eclipse con generación de código

Jonathan Trujillo Bachiller

MAYO 2013

Resumen

El proyecto presentado en esta memoria corresponde al Trabajo Fin de Grado “Un editor gráfico de modelos en Eclipse, con generación de código".

En este documento se describe, como objetivo principal del trabajo, el proceso completo de construcción de un editor gráfico, para diseñar transformaciones de modelos, basado en una línea de investigación que sienta la base de necesidad de este proyecto. De manera complementaria, se define la habilidad de generación de código de la transformación a partir de un diagrama diseñado, para algún lenguaje de transformaciones existente, en nuestro caso Epsilon Generation Language (EGL).

Esta propuesta de la que surge el proyecto, sigue el paradigma de la Ingeniería dirigida por modelos (MDE), basada en trabajar a alto nivel con modelos hasta su correcta definición, que finalmente generará el software automáticamente. Esta investigación, cubre la ausencia y necesidad de lenguajes de modelado para transformaciones, codificados a mano hasta este punto, lo que repercute en la calidad final de las transformaciones porque no se sigue una fase de análisis y diseño previa.

En estas circunstancias, definen un lenguaje de reglas, conocido como transML, para el modelado de transformaciones de modelos. Esta propuesta incluye tanto los elementos del lenguaje de modelado como su representación gráfica, para la que no existe, en la actualidad, un editor gráfico, lo que complica la definición de diagramas y su comprensión por parte de cualquier profesional.

Para la realización del proyecto se trabajará bajo entorno Eclipse y Framework Graphiti a fin de lograr una herramienta completa que se puede integrar en la plataforma Eclipse, de cualquier desarrollador para trabajar con ella añadiéndolo como plug-in.

La memoria contiene los pasos seguidos para la realización del proyecto, elección de tecnología, presentación del lenguaje de reglas que origina la posibilidad de este proyecto, análisis, diseño e implementación de editor gráfico y generación de código junto con los resultados obtenidos que apoyan los objetivos planteados para el proyecto.

Palabras clave

Lenguaje de reglas, transML, MDE, Graphiti, plug-in, Eclipse, generación de código, EGL.

Abstract

The project presented in this document is for the Degree Final Project "A graphical model editor in Eclipse, with code generation."

This document describes the development process of a graphical editor to design model transformations, which was the main objective of the project. The editor also allows the generation of the transformation code from the designed diagrams, for an existing transformation language, in particular for the Epsilon Generation Language (EGL).

This project follows the Model-Driven Engineering (MDE) paradigm, which proposes the use of models to conduct the software development process, from which the code of the final application is generated. In this paradigm, model transformation languages are used to manipulate models. However, currently, there are no design notations for transformations, which have to be developed by hand without a previous analysis and design phase. This has a negative impact in the quality of the implemented transformations.

To alleviate these problems, some researchers of the University Autónoma of Madrid and the University of York proposed transML, a family of languages to model model transformations. This proposal includes both the modeling language elements and their graphical representation. However, there is no graphical editor, which makes it difficult the definition and understanding of diagrams by practitioners.

This project has been developed using Eclipse and Graphiti, an Eclipse-based graphics framework that enables rapid development of diagram editors for domain models, with the aim of achieving full tool that can be integrated in the Eclipse Platform as a plug-in.

This document presents the steps followed in this project, the choice of technology, the modelling language for model transformations that is the basis of the project, the different development phases (analysis, design, and implementation of graphical editor and code generator), and the obtained results.

Key words

Rule language, transML, MDE, Graphiti, plug-in, Eclipse, code generation, EGL.

Índice de contenido

Índice de figuras

1. Introducción

En el presente documento, se explicarán los detalles de la realización del TFG realizado.

Para comenzar, veamos el origen de la necesidad que da sentido a la existencia del proyecto y los objetivos marcados para el mismo.

1.1 Antecedentes

La Ingeniería dirigida por modelos (MDE[1]) es un paradigma de desarrollo donde los modelos son el inicio y base fundamental de un desarrollo, generando a partir de ellos el producto final de manera automática, total o parcialmente. De esta forma se persigue trabajar a alto nivel con modelos hasta su correcta definición que finalmente generará el software automáticamente.

En relación al último paso, aparecen las transformaciones de modelo para ocuparse de la generación de producto a partir del modelo definido. Principalmente se ocupan de la conversión de uno o varios modelos en otro u otros modelos, por ejemplo para refinement, refactoring, etc. Para conseguirlo de manera prácticamente automática se deben definir reglas de transformación a modo de mapeo entre los modelos origen y destino. Este tipo de transformación se denomina Modelo-a-Modelo (M2M).

En MDE, las transformaciones rara vez se especifican con un lenguaje de propósito general, como puede ser Java, sino que se suele definir con lenguajes de transformación M2M especialmente adaptados para la tarea de transformar los modelos. Las transformaciones M2M se implementan como software y, como cualquier otro software, necesitan ser analizados, diseñados, implementados y probados.

Algunos ejemplos de estos lenguajes de transformaciones de modelos especialmente adaptados serían QVT [2], ATL [3], Gramáticas de grafos Triple [4] y ETL [5]. El estilo de este tipo de lenguajes está basado en reglas declarativas, donde cada regla tiene normalmente un dominio de origen (modelo de entrada) y un dominio de destino (modelo de salida), y la regla marca cómo realizar la transformación. Dependiendo del lenguaje, es posible tener más de un dominio de entrada y salida. También dependiendo del lenguaje, puede que haya paso de parámetros entre reglas, dependencias de distinto tipo entre reglas (ej. r1 se puede aplicar sólo si antes se ha aplicado r2), condicionales (ej. si se cumple cierta condición, se ejecutará r1, si no, se ejecutará r2), etc.

Actualmente no existen lenguajes de modelado para transformaciones, sino que se codifican directamente a mano, y eso puede repercutir en la calidad final de esas transformaciones porque no se sigue una fase de análisis y diseño previa.

Siguiendo esta falta de un lenguaje de modelado, un grupo de investigadores propone, un lenguaje para el modelado de transformaciones de modelos [6 **transML**], una propuesta que incluye tanto los elementos del lenguaje de modelado como su representación gráfica, para la que no existe, en la actualidad, un editor gráfico, teniendo que diseñar los diagramas en forma de árbol al no poderse construir gráficamente, lo que complica mucho la definición de diagramas y su comprensión por parte de cualquier profesional.

Para la definición de la transformación, permite definir las reglas que sigue la transformación, los dominios que participan y sus parámetros. Además incorpora elementos para el flujo de ejecución de la transformación, que aportan orden, alternativas de ejecución y llamadas implícitas a otras reglas (o bloques de reglas).

1.2 Objetivos del proyecto

El objetivo principal de este proyecto es dar soporte al lenguaje de reglas de transformación de modelo [6 **transML**].

En primer lugar, y como parte central del proyecto, se desea construir un editor gráfico de diagramas para las reglas de transformación definidas, con el objetivo de que un desarrollador pueda diseñar

cómo implementará una transformación. Para ello se deberán implementar dentro del editor cada uno de los elementos que se definen en transML, bajo sus especificaciones de diseño y comportamiento.

En relación al diseño del editor se buscará trabajar con una herramienta que libre la complejidad interna del trabajo con editores gráficos, aportando los cimientos de un diseño y definición de características que puedan extenderse para cubrir las necesidades del proyecto.

En segundo lugar, siguiendo un enfoque de Desarrollo Dirigido por Modelos se generará, en paralelo al diseño en el editor, un esqueleto del código de la transformación utilizando ficheros XMI, compatibles con ecore, que es el standard de facto para el modelado en Eclipse, conforme a lo especificado en el diagrama diseñado. Esta definición será el parámetro de entrada para un lenguaje de transformación que haga posible la realización final de la transformación diseñada.

Finalmente, el editor incluirá una opción para generar el código de la transformación (un esqueleto) a partir del diagrama, en concreto del equivalente en XMI de la transformación diseñada, para algún lenguaje de transformaciones existente.

Por el alcance del proyecto, y el peso del diseño del editor, esta generación de código se definirá de manera parcial, no llegando a interpretar la totalidad de elementos y relaciones del lenguaje de reglas, por lo que tan sólo podrá utilizarse con diagramas formados por los elementos que se interpreten en la generación de código.

1.3 Estructura del documento

El presente documento contiene toda la información referente al proyecto realizado, siguiendo un orden lógico de acontecimientos.

Así partiendo de los objetivos ya expuestos, en los próximos puntos se verá en detalle la acometida del problema y sus resultados:

- En la sección 2, Estado del arte / tecnologías a utilizar, se desglosarán las tecnologías relacionadas con el ámbito del proyecto y se explicará la elección de algunas de ellas para la realización del mismo.

- En la sección 3, Lenguaje de reglas: (transML) , se explicará el lenguaje de reglas de transformación del que surge la idea de generar un editor de diagramas como centro del proyecto realizado.

- En la sección 4, Análisis, se verá las especificaciones de las partes del editor gráfico en relación a los elementos del lenguaje de reglas, haciendo hincapié en propiedades editables, reglas de diseño concretas y representación gráfica (pictograma) asignada a cada elemento del diagrama. Además se mencionará el planteamiento seguido para la generación de código.

- En la sección 5, Diseño, se verá el diseño seguido en la implementación que se encuentra totalmente marcado por el Framework Graphiti.

- En la sección 6, Implementación, se comentarán puntos generales y algunos más en detalle sobre la implementación realizada.

- En la sección 7, Resultados, se mostrarán resultados del editor creado tanto de figuras individuales que representan los elementos del lenguaje de transformación, así como diagramas completos generados con el editor. A su vez, se verán ejemplos de generaciones de código resultantes de la interpretación de diagramas construidos con el editor.

- En la sección 8, Conclusiones, se expondrán unas breves conclusiones del trabajo realizado y se hablará de opciones de continuación del proyecto.

- El último capítulo del documento, Bibliografía, contiene referencias a documentación utilizada y consultada durante la construcción del proyecto.

2. Estado del arte / tecnologías a utilizar

En este punto se comenta la tecnología relacionada con el proyecto realizado y se indicará la elección final de tecnologías bajo la que se ha desarrollado el proyecto.

2.1 Base de desarrollo

Un entorno de desarrollo integrado (IDE), es un entorno de programación que proporciona un marco de trabajo compuesto por un conjunto de herramientas que facilitan el desarrollo. Compuestos básicamente por un editor de texto, un compilador, intérprete y un depurador. A continuación, veremos los IDE principales relacionados con el ámbito del proyecto:

- Eclipse [7], entorno de desarrollo para el proyecto (Plug-in Project).

  Eclipse es un entorno de desarrollo integrado, de código abierto y multiplataforma. Fue creado originalmente por IBM pasando posteriormente a la Fundación Eclipse, una organización independiente sin ánimo de lucro que fomenta una comunidad de código abierto y una serie de servicios disponibles.

  Dispone de un Editor de texto con resaltado de sintaxis. La compilación es en tiempo real. Hace uso de JUnit para pruebas unitarias, ofrece control de versiones con CVS, integración con Ant, asistentes para creación de proyectos y todo tipo de elementos. Además permite desarrollar con múltiples lenguajes, siendo los principales Java y C.

  El IDE de Eclipse permite mediante plug-in añadir funcionalidad a la plataforma de trabajo.

  En este proyecto se han añadido los plug-in siguientes:

  - Graphiti
  - Epsilon

- EMF (8 Eclipse Modeling Framework), tecnología de modelado.

  Tecnología de modelado que forma la base en Eclipse para el desarrollo dirigido por modelos, siendo útil para el desarrollo de metamodelos, y permite a partir de los metamodelos generar automáticamente las clases en código.

  Aporta un diseño previo, que una vez concretado, genera código completamente funcional y sin errores, basado en nuestro modelo.
  Estas clases java se desarrollan a partir de un modelo de datos en UML, XML (modelo Schema), Java annotations o Ecore (modelo propio de EMF).

  EMF proporciona un marco para guardar la información del modelo, utilizando XMI (XML Metadata Interchange) como formato.

  Creado el meta-modelo utilizando EMF es posible crear las clases Java correspondientes a este modelo, ofreciendo la posibilidad de que el código generado puede ser extendido de forma segura.

2.2 Herramientas para la creación de editores gráficos

Conjunto de herramientas que proporcionan una infraestructura para el desarrollo de editores gráficos, facilitando el proceso y eliminando la alta complejidad interna de este tipo de implementaciones. A continuación se revisan las más utilizadas:

- Graphiti [9] herramienta base sobre la que se ha desarrollado el editor de diagramas.

  Apoya el desarrollo de editores de diagrama para modelos de dominio. Diseñado como un plug-in para Eclipse a partir de la versión 3.4 del IDE, se encuentra enmarcado dentro del área de

modelado de Eclipse, relacionado con Graphical Modeling Framework (GMF) y Graphical Editing Framework (GEF).

La compañía SAP creó esta herramienta para facilitar y ocultar la complejidad interna de cara a desarrollar editores gráficos. Finalmente la donó a la comunidad Eclipse en 2009, dando libre acceso. Es una herramienta estable que ya en SAP trabajaba en producción.

Con Graphiti es posible aprender rápidamente a utilizarlo, aportando además implementaciones por defecto para las características de las figuras. Con esto se consiguen además editores similares al desarrollar con una base tan sólida, extensible, que marca el desarrollo del editor.

- GMF ([10] Graphical Modeling Framework).

Proporciona infraestructura evolucionando a partir de Eclipse Modeling Framework (EMF) para la generación de editores gráficos en eclipse. Sigue un enfoque dirigido por modelos para la creación de diagramas. Ofrece un conjunto de componentes e infraestructura para el desarrollo de editores gráficos basados en EMF, como plug-ins de eclipse.

Para crear un editor gráfico, GMF sigue seis pasos o definiciones de ficheros:

1) Domain model: el metamodelo en que se basa para crear el editor gráfico.
2) Domain Gen Model (.genmodel): se utiliza para generar el código del modelo de dominio con EMF.
3) Graphical Def Model(.gmfgraph): se utiliza para definir los elementos gráficos para el modelo de dominio.
4) Tooling Def Model (.gmftool): se utiliza para definir la paleta de herramientas que se pueden utilizar en el editor gráfico.
5) Mapping Model (.gmfmap): vincula el modelo de dominio, el modelo gráfico (.gmfgraph) Y el modelo de herramientas (.gmftool).
6) Diagram Editor Gen Model (.gmfgen): se utiliza para generar el editor GMF gráfico además de el código EMF generado por el archivo genmodel.

- Eugenia [[11]], herramienta para generar los elementos necesarios para trabajar con GMF y facilitar su uso.

Eugenia automatiza la generación de los modelos gmfgraph, gmftool y gmfmap de GMF, a partir de metamodelos ecore convenientemente anotados. De esta forma, es posible generar editores basados en GMF de una manera más rápida y sencilla. Sin embargo, Eugenia sólo te proporciona una versión inicial del editor, que luego el desarrollador tiene que pulir usando GMF.

- DSL Tools, herramienta de creación y explotación de gráficos.

Un lenguaje específico de dominio (DSL) es un lenguaje de programación creado para dar solución a un problema de un dominio concreto.
En el alcance que nos interesa, se presentan entonces una serie de herramientas dentro de Eclipse, un conjunto de herramientas para trabajar con modelos Ecore: un editor gráfico, generación de código Java para aplicaciones RCP y relacionadas con EMF, soporte para comparación de modelos, apoyo a esquemas XSD, tiempos de ejecución, OCL y modelador gráfico. Incluye un SDK completo, herramientas de desarrollo y código fuente.

2.3 Lenguaje para la generación de código

Los lenguajes de generación de código permiten generar código basándose en modelos. Por medio de plantillas, escritas en diversos lenguajes según el soporte utilizado, se interpreta el modelo diseñado y se consigue su transformación a código. A continuación veremos algunos lenguajes:

- Epsilon Generation Language (EGL) [[12]], lenguaje para generación de código a partir de un modelo.

El lenguaje está construido sobre EOL, un lenguaje para la manipulación de modelos con una sintaxis mezcla de OCL (ej. para consultas sobre colecciones) y Javascript (ej. para la definición de variables, bucles, etc.). EGL, además, proporciona algunas características que simplifican la generación de texto desde modelos, incluyendo: regiones protegidas para evitar la sobreescritura de código previamente generado, algoritmos para formatear el texto generado, y mecanismos de trazabilidad.

- Acceleo [13], soporte para generación de código.

  Es un generador de código abierto de la Fundación Eclipse que permite a los desarrolladores basarse en modelos para crear la aplicación. Se utiliza para la transformación de modelo a texto. Está escrito en Java y se implementa como un plug-in en el IDE Eclipse. Además, proporciona herramientas para la generación de código a partir de modelos basados en EMF.

- Jet [14], apoyo mediante plantillas y etiquetas para la generación correcta de código.

  Contenido dentro de EMF. En JET (Emisor de plantillas Java), se utiliza un subconjunto de sintaxis JSP que facilita la creación de plantillas para expresar el código que se desea generar. JET posee un motor de plantilla genérica que puede utilizarse para generar SQL, XML, fuentes Java y otros tipos de salida a partir de plantillas.

Finalmente, las **tecnologías utilizadas en el proyecto** han sido las siguientes:

- **Eclipse**, como base de desarrollo porque existen muchas herramientas de modelado para Eclipse, y es una herramienta de uso extendido y gratuita. Hacer el editor para Eclipse permite integrarlo con otras herramientas existentes también basadas en EMF.

  Además de utilizarlo para desarrollar, el editor implementado será un plug-in para Eclipse, una herramienta que se podrá añadir a Eclipse y ser utilizada desde dicho entorno por cualquier usuario.

- **EMF**, utilizado como marco para guardar la información del modelo, utilizando XMI. El editor diseñado tendrá su correspondiente interpretación en formato XMI que podrá ser leído e interpretado por el lenguaje de generación de código para realizar la transformación final.

- **Graphiti**, como herramientas para la creación de editores gráficos porque es la herramienta más flexible y con menor curva de aprendizaje de todas las que aparecen en esa lista, y permite personalizar mucho los entornos que se construyen con él.

- **Epsilon**, como lenguaje de generación de código, por elección personal, ya que lo cierto es que podría haberse usado cualquiera de los lenguajes comentados.

3. Lenguaje de reglas: (transML)

Lenguaje para el modelado de transformaciones de modelos que incluye tanto los elementos del lenguaje de modelado como su representación gráfica [Sección 4]

Es independiente del lenguaje utilizado para realizar la transformación y permite modelar, mediante la definición de estructuras de reglas, que pueden ser unidireccionales o bidireccionales, si la transformación se puede realizar origen-destino y viceversa.

Para la definición de la transformación, permite definir las reglas que sigue la transformación, los dominios que participan y sus parámetros. Además incorpora elementos para el flujo de ejecución de la transformación, que aportan orden, alternativas de ejecución y llamadas implícitas a otras reglas (o bloques de reglas).

En este punto se muestra la definición y representación en XMI del modelo de domino de los elementos del diagrama de reglas.

En la definición del modelo de dominio [Imagen 1] podemos ver cómo se relacionan los elementos del dominio y las restricciones del modelo que marcarán su implementación (dónde se pueden añadir, lo que conlleva su borrado, etc.).

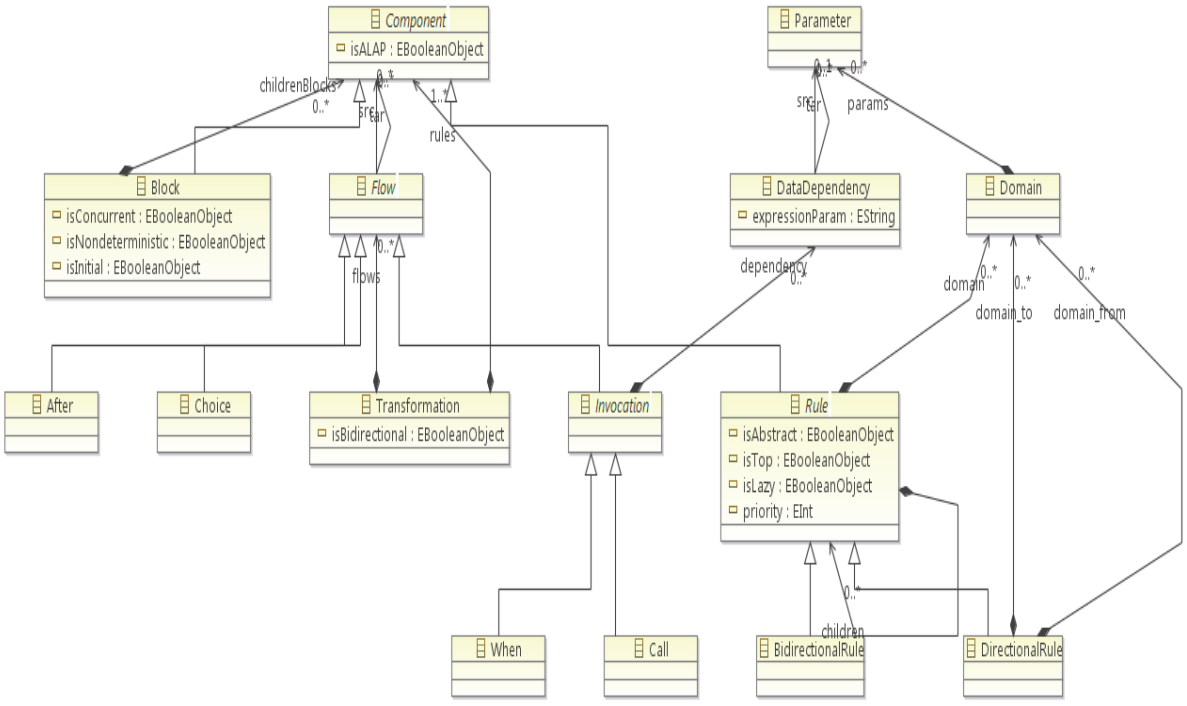


Imagen 1 : transML - Definición del modelo de dominio

Del diagrama anterior podemos extraer que una transformación está formada por reglas (Rule), donde cada regla tiene dominios definidos (Domain), y estos dominios pueden tener una serie de parámetros (Parameter).

Por otro lado, existen relaciones (Flow) que marcan el orden de ejecución de las reglas, existiendo un tipo de relación que invoca (Invocation) a otra regla pudiendo indicar una dependencia (DataDependency) por medio de un parámetro si se desea.

Además todos estos componentes se pueden enmarcar dentro de bloques (Block) para trabajar de manera agrupada.

Veamos de manera detallada qué significa cada elemento, atributos y reglas de validación.

3.1 Clase Transformation a nivel de Diagrama

Representa una transformación en XMI.

❖ Objeto de dominio

Aunque no se representa gráficamente, tiene que haber un objeto de este tipo en cada diagrama, el cual contendrá la definición de todas las reglas, bloques, etc. del diagrama.

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" ...>
<pi:Diagram ...>
</pi:Diagram>
<rule:Transformation isBidirectional=”false”>
        <rules xsi:type="rule:Block" ...>
            ...
        </rules>
</rule:Transformation>
</xmi:XMI>
```

❖ Atributos

- **isBidirectional** : si es true, la transformación puede ejecutarse en dos direcciones: de origen-a-destino y de destino-a-origen; si es false, la transformación puede ejecutarse únicamente en una dirección: origen-a-destino.

3.2 Clase Component

Es una clase abstracta, su representación está definida para sus subclases (Block y Rule).

❖ Atributos

- **isALAP**: viene de "as long as possible", y significa que la regla o bloque de reglas se ejecutará tantas veces como sea posible, si es true, o una sola vez, si es false.

3.2.1 Clase Block

Permite representar un bloque o conjunto de componentes relacionados.

❖ Objeto de dominio

Si el bloque se dibuja directamente en el diagrama, fuera de otros bloques, el objeto de dominio se añadirá al atributo rules del objeto Transformation asociado al diagrama:

```
<Transformation...>
        <rules Block.../>
</Transformation>
```

Si el bloque se dibuja dentro de otro bloque, no se añadirá al atributo rules del objeto Transformation, sino al atributo childrenBlocks del bloque contenedor.

```
<Block ...>
        <childrenBlocks Block.../>
</Block>
```

❖ Atributos

- **isConcurrent**: significa que las reglas dentro del bloque pueden ejecutarse de manera concurrente, si es true, o sólo puede ejecutarse una regla a la vez si es false.
- **isNondeterministic**: si es true, significa que el orden de ejecución de las reglas dentro del bloque es no determinista (es decir, aleatoria), mientras que si el atributo es false, el orden vendrá dado por su prioridad, o por relaciones de flujo when, call, etc.
- **isInitial**: identifica la primera regla a ejecutar dentro del bloque.

3.2.2 Clase Rule

Es una clase abstracta, su representación está definida para sus subclases (DirectionalRule y BidirectionalRule).

La clase DirectionalRule representa una regla unidireccional de la transformación. La clase BidirectionalRule representa una regla bidireccional de la transformación. La representación es la misma que para reglas direccionales, así como el modo en que se maneja el objeto de dominio.

❖ Objeto de dominio

Si la regla se dibuja directamente en el diagrama, fuera de un bloque, el objeto de dominio se añadirá al atributo rules del objeto Transformation asociado al diagrama:

```
<Transformation...>
        <rules DirectionalRule.../>
</Transformation>
```

Si la regla se dibuja dentro de un bloque, no se añadirá al atributo rules del objeto Transformation, sino al atributo childrenBlocks del bloque contenedor.

```
<Block ...>
        <childrenBlocks DirectionalRule.../>
</Block>
```

❖ Atributos

- **isAbstract**: la regla es abstracta.
- **isTop**: es una construcción típica de QVT, e identifica cuáles son las reglas que se evaluarán en primer lugar.
- **isLazy**: es una regla que no invoca el motor de transformación, sino que siempre se invoca desde otra regla.
- **Priority**: establece una prioridad en el orden de ejecución de las reglas, de manera que siempre se ejecutarán primero las que tengan prioridad 1, luego las que tengan prioridad 2, etc.

3.3 Clase Domain

Dominio de la regla, que puede ser de entrada o de salida. Cada regla puede tener un número arbitrario de dominios de entrada y salida.

❖ Objeto de dominio

El objeto Domain se guarda en el atributo domain_from o domain_to de la regla a la que se asocia, dependiendo de si es un dominio de entrada o de salida de la regla.

```
<Transformation...>
        <rules DirectionalRule...>
                <domain_from name=".." languague="…"/>
                <domain_to name=".." languague="…"/>
        </rules>
</Transformation>
```

3.4 Clase Parameter

Representa un parámetro de una regla, y van asociados a un dominio de la regla.

❖ Objeto de dominio

El objeto Parameter se guarda dentro del atributo params del dominio al que se asocia.

```
<Transformation...>
    <rules DirectionalRule...>
        <domain_from name=".." languague="…">
            <params name="…">
                <type type="…" href="…"/>
            </params>
        </domain_from>
    </rules>
</Transformation>
```

3.5 Clase After

Indica que un componente (regla o bloque) se ejecuta después de otro.

❖ Objeto de dominio

Se crea un objeto de tipo After, y se asigna correctamente los atributos src (objeto origen de la flecha) y tar (objeto destino de la flecha). Aunque los atributos src y tar son de tipo lista, sólo habrá un objeto en cada uno.

```
<Transformation ...>
    <rules ...>
    <flows After src="..." tar="..."/>
</Transformation>
```

3.6 Clase Call

Indica que un componente (regla o bloque) llama a otro.

❖ Objeto de dominio

Se crea un objeto de tipo Call, y se asigna correctamente los atributos src (objeto origen de la flecha) y tar (objeto destino de la flecha). Aunque los atributos src y tar son de tipo lista, sólo habrá un objeto en cada uno.

```
<Transformation ...>
    <rules ...>
    <flows Call src="..." tar="..."/>
</Transformation>
```

3.7 Clase When

Indica que un componente (regla o bloque) se ejecuta cuando se ha ejecutado otro.

❖ Objeto de dominio

```
<Transformation ...>
    <rules ...>
    <flows When src="..." tar="..."/>
</Transformation>
```

3.8 Clase DataDependency

Representa el paso de un parámetro en una llamada entre componentes (reglas o bloques).

❖ Objeto de dominio

Se crea un objeto de tipo DataDependency dentro de la llamada, y se asigna correctamente su atributo tar (parámetro destino de la flecha).

```
<flows Call src=”...” tar=”...”/>
        <dependency expressionParam=”” src=”” tar=””>
</flows>
```

3.9 Clase Choice

Indica que un componente (regla o bloque) llama a otro, si se cumple una condición.

❖ Objeto de dominio

Se crea un objeto de tipo Choice, y se asigna correctamente los atributos src (objeto origen de la flecha) y tar (objeto destino de la flecha). Si se añaden nuevos componentes de entrada o salida al Choice, simplemente hay que añadirlos al atributo src o tar, según corresponda. Además, se compone de un atributo para indicar la restricción.

```
<Transformation ...>
        <rules ...>
        <flows Choice src=”...” tar=”...”>
                <constraint OpaqueConstraint text=”...” />
        </flows>
</Transformation>
```

4. Análisis

En este punto veremos el análisis de las partes del editor gráfico, señalando la forma de trabajar con cada una dentro del editor.

El editor estará formado por una barra de herramientas lateral donde se muestran entradas para cada elemento del lenguaje. Así, pulsando en un elemento de la barra de herramientas y arrastrándolo al canvas del editor, o sobre algún otro componente del diagrama como veremos, se añade el elemento al diagrama.

A continuación veremos en detalle ciertas especificaciones, destacables de cada elemento del diagrama, por su complejidad o diferencia frente al resto de componentes del diagrama, como son las propiedades editables, reglas de layout concretas y representación gráfica (pictograma) asignada a cada elemento del diagrama.

4.1 Diagrama

El diagrama se crea directamente al ser un diagrama de Graphiti, eligiendo como tipo de diagrama el generado en el proyecto actual lo que le aporta los objetos y funcionalidades aquí estudiadas. Para conseguirlo es preciso tener instalado en Eclipse el plug-in desarrollado.

Directamente relacionado con el diagrama se encuentra el elemento Transform. El editor no incluirá un botón para crear este tipo de objeto, sino que se creará por defecto la primera vez que se añada un elemento al mismo

- Hoja de propiedades

Precisa hoja de propiedades para manejar la propiedad isBidirectional (por defecto a false). Si es true, no se pueden añadir al diagrama DirectionalRules. Si es false, no se pueden añadir al diagrama BidirectionalRules. Si se añaden elementos al diagrama, el atributo dejará de ser editable.

- Pictograma

No tiene una representación gráfica. Su representación será en fichero XMI, del diagrama, conteniendo la información correspondiente del dominio.

4.2 Clase Block

- Reglas de layout

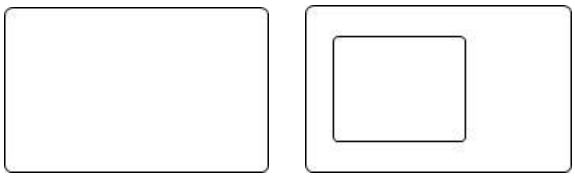
Dentro de un bloque se pueden añadir otros bloques, reglas direccionales y reglas bidireccionales. Al mover un bloque, se tienen que mover todos los componentes que tiene dentro.

- Hoja de propiedades

Precisa hoja de propiedades para editar los atributos isConcurrent, isNondeterministic, isAlap e isInitial de un bloque.

- Pictograma

Una caja transparente con los bordes redondeados.



4.3 Clase Rule

❖ Reglas de layout

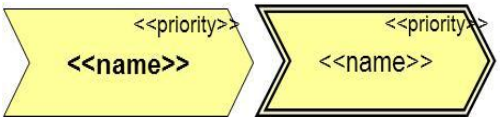
Si el atributo isTop de la regla es true, el borde de la regla será doble. Si el atributo priority de la regla es mayor que 0, el valor del atributo se mostrará en la esquina superior derecha de la flecha.

❖ Hoja de propiedades

Precisa hoja de propiedades para editar los atributos isAbstract, isTop, isAlap, isLazy y priority de una regla.

❖ Pictograma

Una "flecha" amarilla, con el nombre de la regla dentro. Se indica valor priority si es mayor que cero y doble línea según valor atributo Top.



4.4 Clase Domain

❖ Reglas de layout

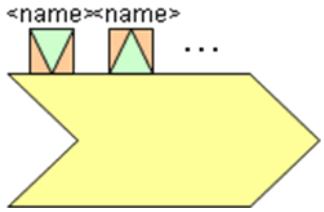
Sólo se pueden crear "dentro" de reglas (es decir, al arrastrar sobre el diagrama se debe pinchar sobre una regla). Se muestran adyacentes a la regla. Se añaden por defecto en la parte superior izquierda de la regla. Se pueden mover alrededor de la regla dentro de un rectángulo imaginario que obliga a estar próxima a la regla para facilitar la interpretación visual del diagrama.

❖ Hoja de propiedades

Precisa hoja de propiedades para editar el atributo name y el atributo language mediante un combo que debe contener los meta-modelos registrados en Eclipse. Al cambiar el Language de un dominio, deben borrarse los tipos de sus parámetros.

❖ Pictograma

Un cuadrado naranja con un triángulo verde, y el nombre del dominio encima o dentro del dominio. La dirección del triángulo indica si el dominio es de entrada (punta hacia abajo) o de salida (punta hacia arriba).



4.5 Clase Parameter

❖ Reglas de layout

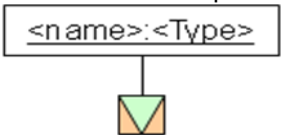
Sólo se pueden crear "dentro" de dominios (es decir, al arrastrar sobre el diagrama se debe pinchar sobre un dominio).

❖ Hoja de propiedades

Precisa hoja de propiedades para indicar el tipo del parámetro. En un combo se cargan las clases definidas en el meta-modelo seleccionado en el dominio al que pertenece dicho parámetro.

❖ Pictograma

Se visualiza como una caja blanca, conectada con una línea al dominio asociado. Dentro de la caja se muestra el nombre del parámetro + ':' + tipo del parámetro, subrayados.



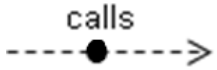
4.6 Clase Call

❖ Reglas de layout

Para dibujarla, se seleccionará como origen un bloque o regla, y como destino otro bloque o regla (también puede ser el mismo). Puede haber varios Call entre los mismos componentes.

❖ Pictograma

Como una flecha con el texto “calls” y un círculo negro sobre la flecha.



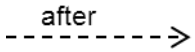
4.7 Clase After y When

❖ Reglas de layout

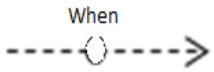
Para dibujarla, se seleccionará como origen un bloque o regla, y como destino otro bloque o regla (también puede ser el mismo). No puede haber dos flechas “after” entre los mismos objetos. Para ello, no debe dejar seleccionar como destino un componente si ya está conectado al componente seleccionado como origen.

❖ Pictograma

Para After se dibujará como una flecha con el texto “after”.



El pictograma de When es igual que el de la clase Call [ver Clase Call] pero el círculo es de color blanco, y encima de la flecha debe mostrarse el texto “when”. El comportamiento y la forma de crear el objeto de dominio es igual al de la clase After [ver Clase After].



4.8 Clase DataDependency

❖ Reglas de layout

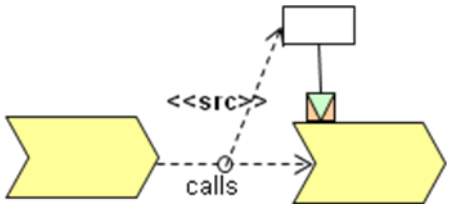
Para dibujarla, se seleccionará como origen un Call o un When, y como destino un parámetro. Se tiene que comprobar que el parámetro está conectado a un dominio de la regla de destino

❖ Hoja de propiedades

Precisa hoja de propiedades para editar el atributo src de una dependencia. Se creará una hoja de propiedades donde habrá una lista desplegable con los parámetros definidos en la regla de llamada. Se podrá elegir uno de ellos. El nombre del parámetro seleccionado se mostrará sobre la flecha. También habrá un campo de texto para editar el atributo expressionParam, que se mostrará si no hay parámetro de entrada seleccionado.

❖ Pictograma

Como una flecha, con el nombre del parámetro de entrada encima.



4.9 Clase Choice

❖ Reglas de layout

Para dibujarla, se seleccionará como origen un bloque o regla, y como destino otro bloque o regla (también puede ser el mismo). Puede haber varios Choice entre los mismos componentes (es decir, no hay que comprobar nada).

Un Choice puede tener varios componentes de entrada y varios componentes de salida, todos compartiendo la misma restricción. Para añadir un nuevo componente de entrada, seleccionar el componente y luego el Choice. Para añadir un componente de salida, seleccionar el Choice y luego el componente.

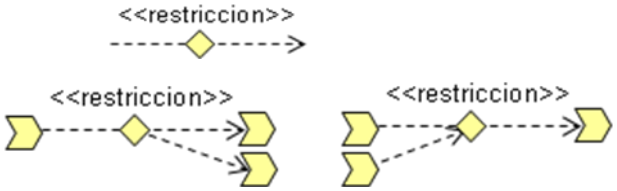
❖ Hoja de propiedades

Precisa hoja de propiedades para editar la restricción mediante un campo de texto. El texto se mostrará sobre la flecha.

```
<Transformation ...>
        <rules ...>
        <flows Choice src="..." tar="...">
                <constraint OpaqueConstraint text="..." />
        </flows>
</Transformation>
```

❖ Pictograma

Como una flecha con un rombo amarillo encima, y el texto de la restricción. Puede tener varias entradas y salidas.



4.10 Generación de código

Esta parte del proyecto se encargará de la generación de un código en lenguaje ETL, generado a partir de la interpretación del diagrama mediante el uso de plantillas EGL creadas.

Para ello, se definirá una plantilla EGL que se invoca desde una clase, pasándole el modelo de dominio, obtenido del diagrama, con el que trabajará la plantilla y el directorio de salida, indicado por el usuario.

La plantilla de salida base será la siguiente para cada modelo de transformación, siguiendo la sintaxis concreta para una regla de transformación definida en el libro del lenguaje Epsilon [15]:

```
// Rationale
(@abstract)?
(@lazy)?
rule  <name>
transform   <sourceParameterName>:<sourceDomainName>!<sourceParameterType>
to      (<rightParameterName>:<rightDomainName>!<rightParameterType>
        (, <rightParameterName>:<rightDomainName>!<rightParameterType>)*
 (extends (<ruleName>)? {  }
```

En relación al lenguaje ETL, se definirán una serie de validaciones sobre el diagrama diseñado a fin de comprobar que se puede generar una salida correcta. Estas comprobaciones específicas del lenguaje ETL son que las reglas tienen exactamente un dominio de entrada y uno de salida, cada dominio tiene un parámetro y ambos están tipificados.

❖ Reglas de layout

Para generar una salida a partir del diagrama se deberá crear un menú contextual, pulsar el botón derecho sobre una parte libre del diagrama, opción Acciones del menú contextual y dentro de la misma una acción para generación de código que realice el proceso.

5. Diseño

En este punto veremos la base del diseño del aplicativo que ha sido guiado por los principios marcados en el Framework Graphiti, que veremos en detalle.

Además se comentará la estructura del proyecto creado y diseños base para el desarrollo.

5.1 Framework Graphiti

Como se comentó en los objetivos del proyecto [Sección 1.2], se parte de la necesidad de dar soporte a una familia de lenguajes para el diseño de transformaciones que potencie su uso mediante la creación de diagramas de transformación correctos para el dominio que se maneja.

Esta tarea se considera la **primera parte del proyecto** y consiste en la completa creación de un editor para el manejo de todos los elementos y atributos descritos en el lenguaje de reglas [Sección 3].

Si se trata de construir un editor gráfico para un modelo de dominio utilizando Eclipse, aparecen los conceptos Eclipse Modeling Framework (EMF) y el marco para editar los gráficos (GEF). Mientras EMF proporciona la base para el modelado, GEF es compatible con la programación de editores gráficos.

Esto es algo que tiene una curva de aprendizaje bastante alta, por lo que para atacar este proyecto es necesario un marco de desarrollo que libre de estas dificultades para poder centrar el esfuerzo en la construcción de un editor gráfico directamente.

SAP creó **Graphiti** [9], una herramienta para facilitar y ocultar esta complejidad interna de cara a desarrollar editores gráficos. Esta herramienta finalmente la donó a la comunidad Eclipse teniendo libre acceso a su uso.

Se decide por tanto desarrollar el editor haciendo uso del plug-in de Graphiti para Eclipse, donde el *diseño de la aplicación está guiado por el framework graphiti implementado en Java*, que marca cómo estructurar las clases de la aplicación, extendiendo las características de las features para cubrir los requisitos que marca el dominio y el lenguaje de reglas.

Es momento de ver en profundidad el trabajo que va a requerir desarrollar sobre Graphiti para conseguir diseñar los elementos del lenguaje. La comunidad Eclipse ofrece un tutorial completo [16] con ejemplos donde se pueden ver las propiedades y habilidades que se pueden manejar para crear el diagrama esperado.

Para desgranar el análisis de extensiones que precisa este proyecto, se hablará por separado del diseño de las representaciones gráficas y por otro lado de las propiedades y habilidades que se deberán extender debidamente.

5.1.1 Representaciones gráficas

Viendo la arquitectura base de Graphiti en torno a la representación final de los elementos [Imagen 2] podemos ver los elementos, que serán necesarios implementar y de qué manera, para que el dominio esté ligado a la representación.

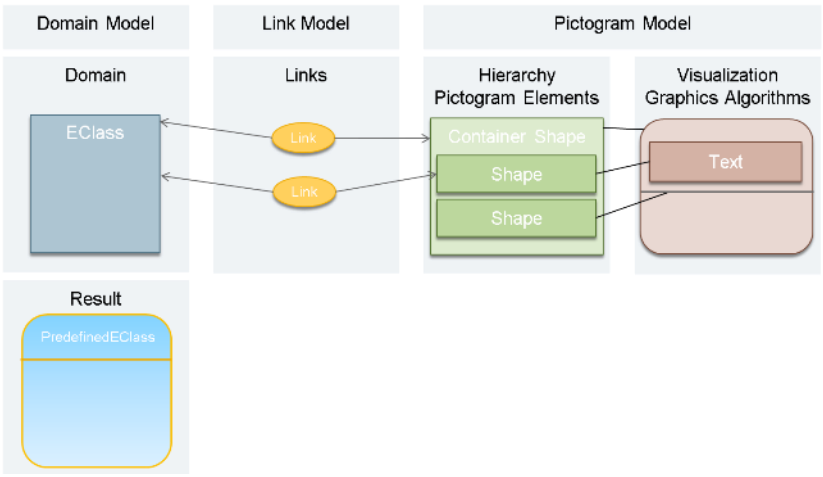


Imagen 2 : Graphiti - Link objetos dominio con representación

Como vemos, cada elemento del dominio estará ligado a un elemento del pictograma, que podrá contener varios elementos a fin de configurar la figura esperada, obteniendo la representación oportuna en cada caso.

5.1.2 Características manejables (features)

Por defecto, al comenzar a trabajar con el framework Graphiti, te proporciona una serie de clases base para cada tipo de acción a realizar con el editor (crear, borrar, etc.), y por cada objeto del dominio que se desee que tenga ese comportamiento o una extensión del mismo, es preciso crearse una clase que herede de una de esas clases base del framework e implementar un conjunto de métodos para lograr el objetivo de diseño y funcionalidad.

Si vemos en detalle la estructura de manejo de features [Imagen 3], podemos ver las características que se van a extender, si es necesario, para cada figura que se maneje. En caso contrario trabajarán con la clase base propia de cada feature:

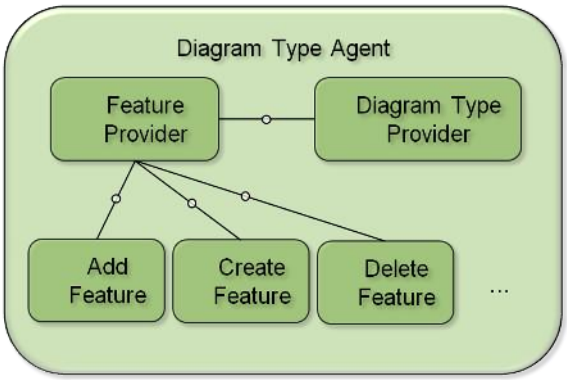


Imagen 3 : Graphiti - Estructura de features

- **Create**
  Se extiende AbstractCreateFeature a fin de definir condiciones en las que se puede crear el objeto de dominio. Así por ejemplo, ciertas figuras podrán crearse únicamente teniendo como figura padre una existente no pudiéndose incorporar al diagrama directamente.

- **Add**
  Se extiende AbstractAddShapeFeature definiéndose en este punto (método add) la representación que tendrá el elemento del dominio como figura dentro del diagrama.

  Como se puede ver en la estructura de ejemplo básica para un elemento [Imagen 4], es imprescindible mantener relacionados los componentes de la figura con el elemento de dominio que representa a fin de poder trabajar posteriormente con la figura creada dentro del contexto que se precisa en cada caso.

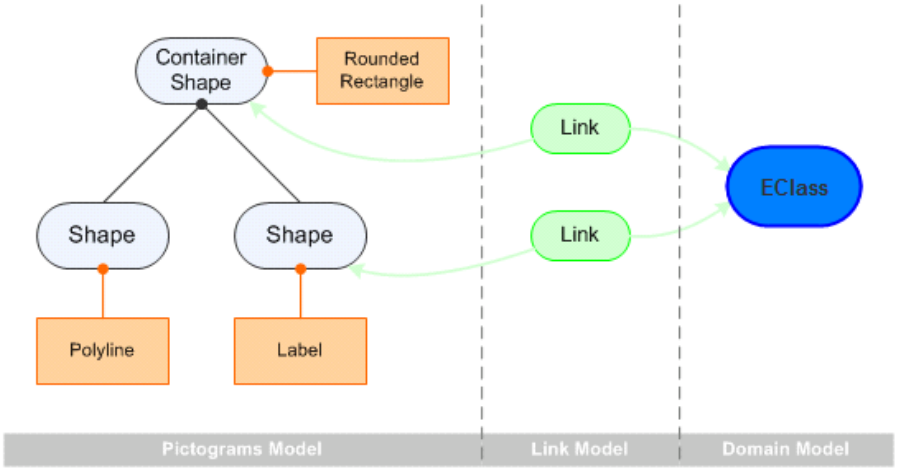


Imagen 4 : Graphiti - Ejemplo relaciones componentes con dominio

Las figuras necesitan tener relación con el objeto de dominio para trabajar con sus atributos a fin de representar su nombre, cambiar la representación según valores de propiedades concretas o manejar cambios en las figuras que deben realizar cambios en dominio.

- **Update**
  Se extiende AbstractUpdateFeature comprobando en este punto primero si una figura necesita revisar su representación (método updateNeeded) y si es así se manejan los cambios necesarios (método update) interpretando las propiedades actuales de dicha figura.

- **Delete**
  Se extiende DefaultDeleteFeature y su finalidad será el control de borrado controlado de las figuras eliminando su relación con el dominio de manera correcta. Se tendrá que tener en cuenta las relaciones propias del modelo de dominio en torno a relaciones que impliquen borrado en cascada de figuras y/o relaciones dependientes de la existencia de otras.

- **Move**
  Se extiende DefaultMoveShapeFeature y se controlará si una figura puede o no moverse, y si puede se controlarán si la posición que va a adquirir es correcta teniendo en cuenta su disposición dentro del modelo. Así por ejemplo si una figura está contenida dentro de un Block, no debe poder moverse fuera del mismo.

- **Layout**
  Se extiende AbstractLayoutFeature y se controlará el redimensionamiento de ciertas figuras, acotando unas medidas mínimas.

- **Properties**
  Además para ciertas figuras será necesario mostrar una hoja de propiedades a fin de poder modificar valores de atributos.
  Cada elemento del diagrama, que lo necesite, deberá tener su clase Filter que extienda AbstractPropertySectionFilter donde se comprueba si una figura es la esperada para mostrar la hoja de propiedades en cuestión. Además se deberá tener una clase Section que extienda GFPropertySection donde se diseña la hoja de propiedades y se manejan los cambios de atributos sobre la misma, traspasando los cambios a los datos del modelo.

5.2 Estructura proyecto

En este punto veremos la estructura creada a nivel de paquetes en el proyecto y veremos la línea de desarrollo seguida mediante la visualización de un diagrama de clases de alto nivel.

Además, se comentarán en profundidad las clases bases desarrolladas para guiar la implementación final.

5.2.1 Estructura de paquetes

Para el desarrollo del proyecto se ha construido una jerarquía de paquetes con coherencia, siguiendo las partes de diseño en las que se descompone el proyecto y aunando en cada paquete funcionalidades comunes.

En un primer nivel [Imagen 5] se ha dividido la estructura en tres bloques principales.

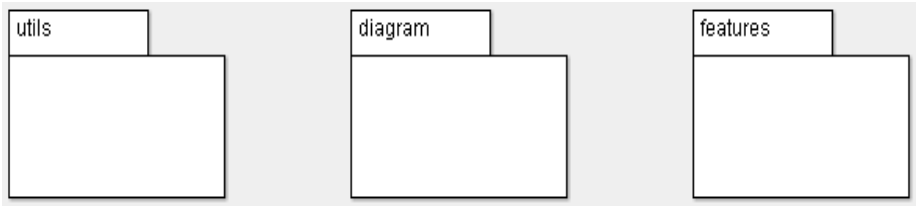


Imagen 5 : Paquetes - Primer nivel paquetes

- Dentro de **Diagram** se encuentran los Providers, puntos de entrada del aplicativo que manejan las funcionalidades disponibles en el sistema.
- Dentro de **Utils** se encuentran una serie de clases de uso general dentro del proyecto como pueden ser mensajes mostrados, funciones de notificación y manejo de propiedades de elementos.

Dentro del paquete **Features** [Imagen 6] aparecen varios subniveles de acuerdo a las partes de desarrollo marcado por Graphiti que ya se ha comentado en puntos anteriores.

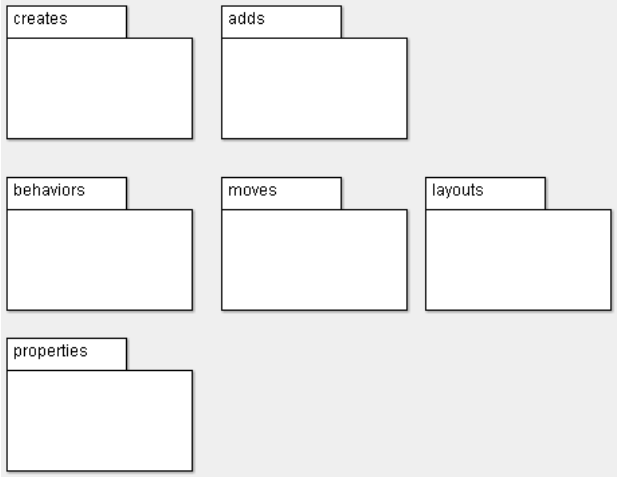


Imagen 6 : Paquetes - Paquetes Features

A nivel del paquete **Creates** [Imagen 7] se define un paquete de clases base que extenderán las conexiones y figuras normales.

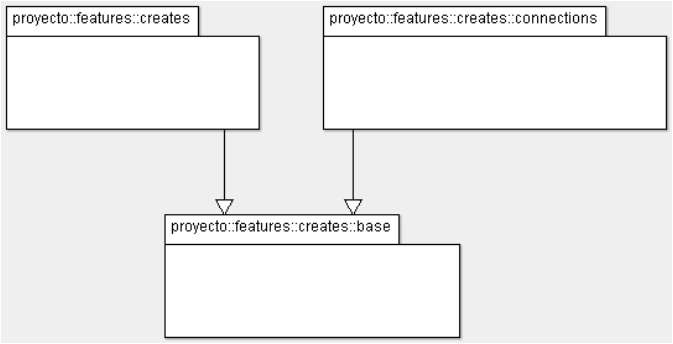


Imagen 7 : Paquetes - Paquetes Creates

A continuación se muestra un diagrama de clases de alto nivel para este nivel de paquetes.

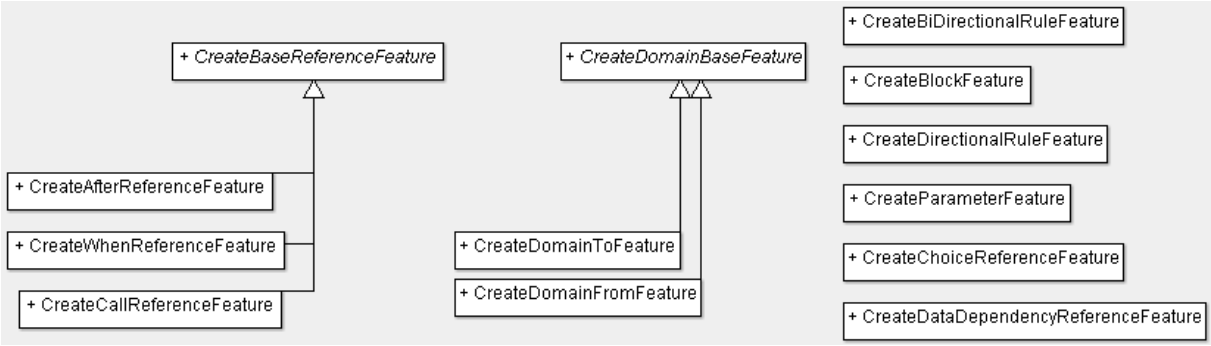


Imagen 8 : Paquetes - Diagrama clases Creates

A nivel del paquete **Adds** [Imagen 9] se define un paquete de clases base que extenderán las conexiones y figuras normales.

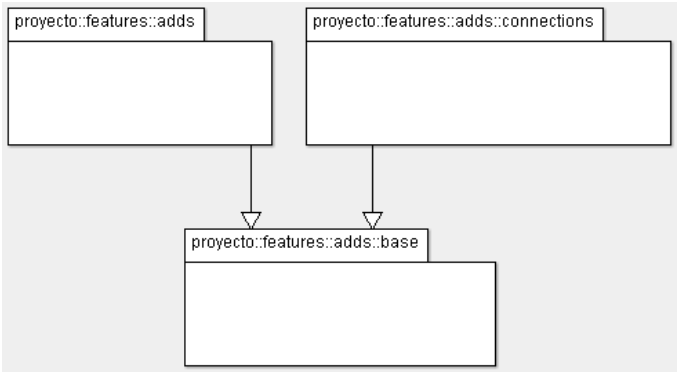


Imagen 9 : Paquetes - Paquetes Adds

A continuación se muestra un diagrama de clases de alto nivel para este nivel de paquetes.

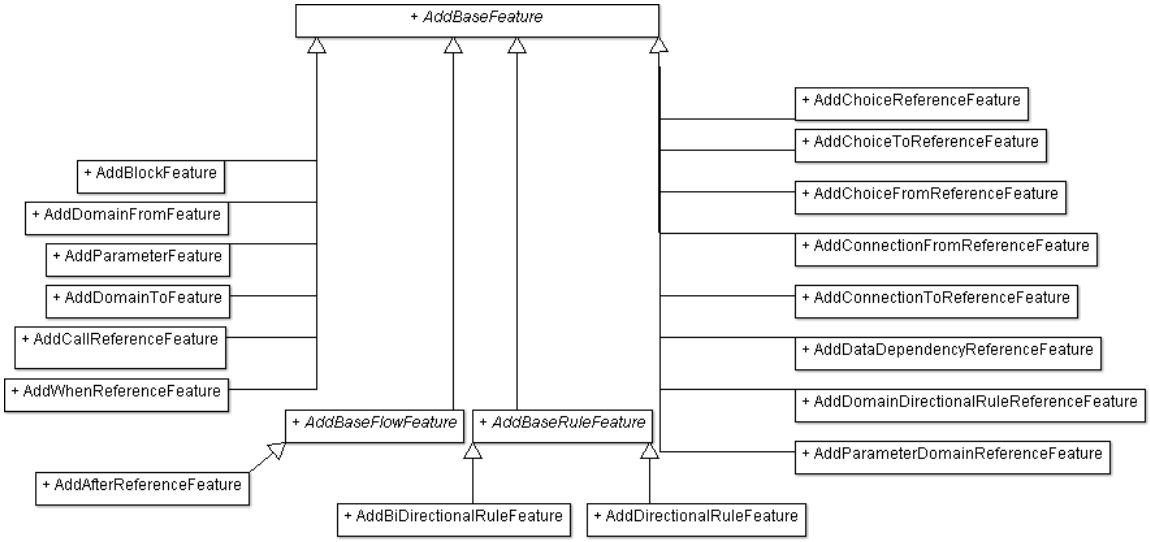


Imagen 10 : Paquetes - Diagrama clases Adds

A nivel del paquete **Behaviors** [Imagen 11] se definen dos subniveles, deletes y updates, que son las habilidades extendidas en el proyecto de manera más concreta para cada figura manejada.

Imagen 11 : Paquetes - Paquetes Behaviors

A continuación se muestra un diagrama de clases de alto nivel para este nivel de paquetes.

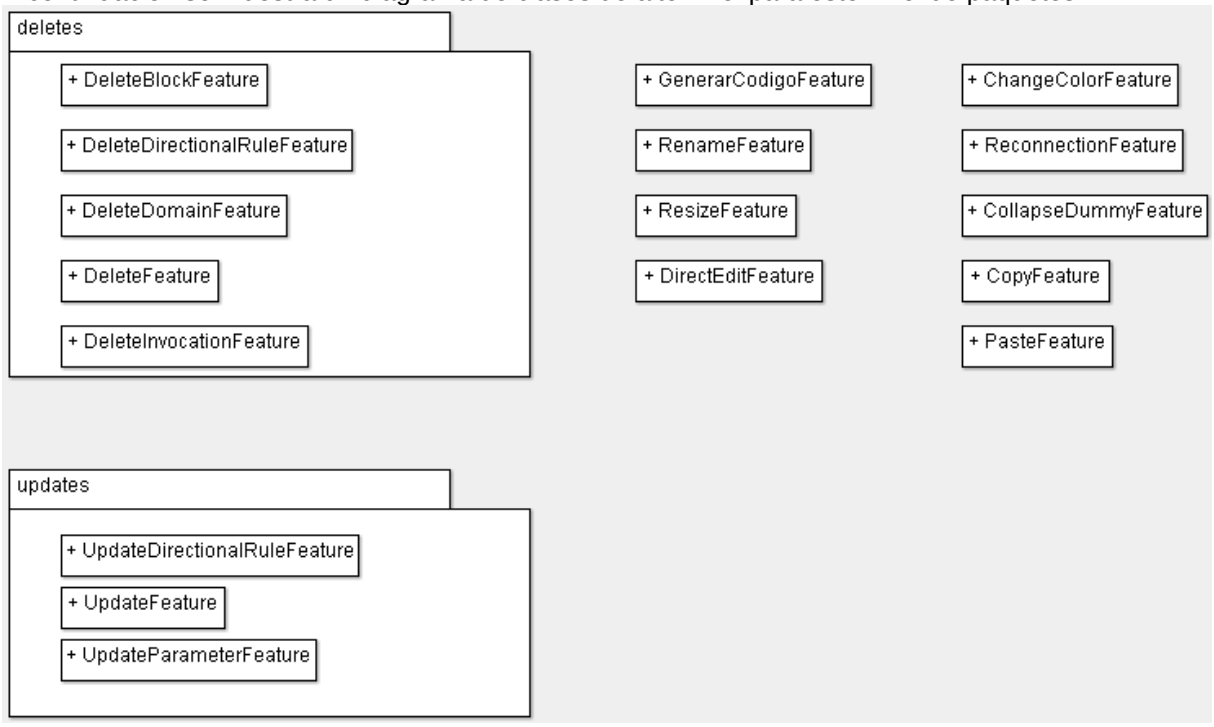


Imagen 12 : Paquetes - Diagrama clases Behaviors

A nivel del paquete **Layouts** [Imagen 13] se define un paquete de clases base que extenderán el resto de layouts definidos.

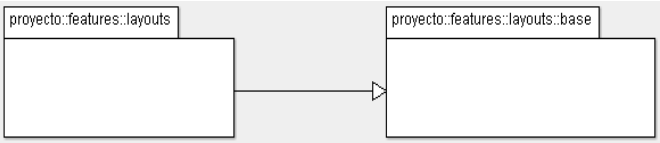


Imagen 13 : Paquetes - Paquetes Layouts

A continuación se muestra un diagrama de clases de alto nivel para este nivel de paquetes.

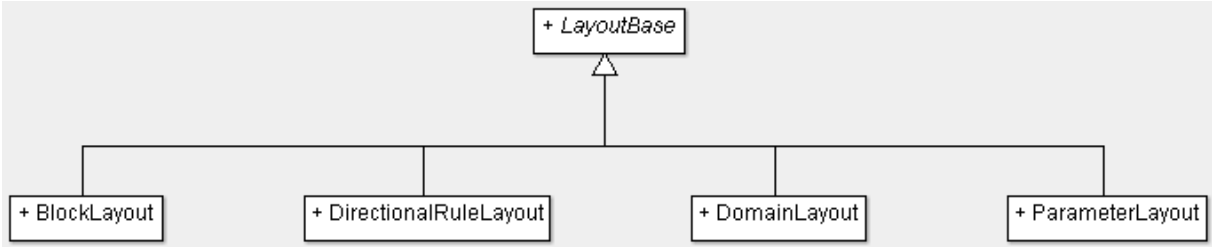


Imagen 14 : Paquetes - Diagrama clases Layouts

A nivel del paquete **Moves** [Imagen 15] se definen clases directamente para control de movimientos específicos.

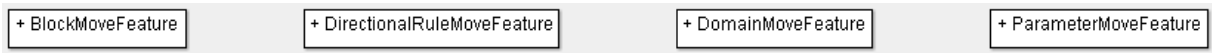


Imagen 15 : Paquetes - Diagrama clases Moves

A nivel del paquete **Properties** [Imagen 16] se separa en Filters (condiciones creación) y Sections (hojas de propiedades). Además, se define a nivel de Sections un paquete de clases base que extenderán las sections creadas y un paquete utils que utilizarán esas mismas sections.

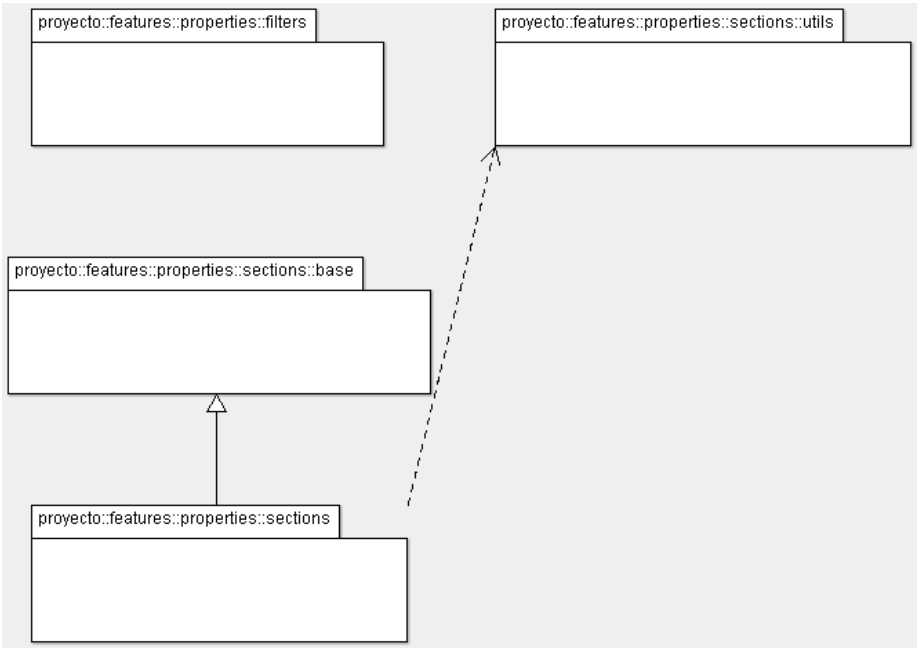


Imagen 16 : Paquetes - Paquetes Properties

A continuación se muestra un diagrama de clases de alto nivel para este nivel de paquetes.

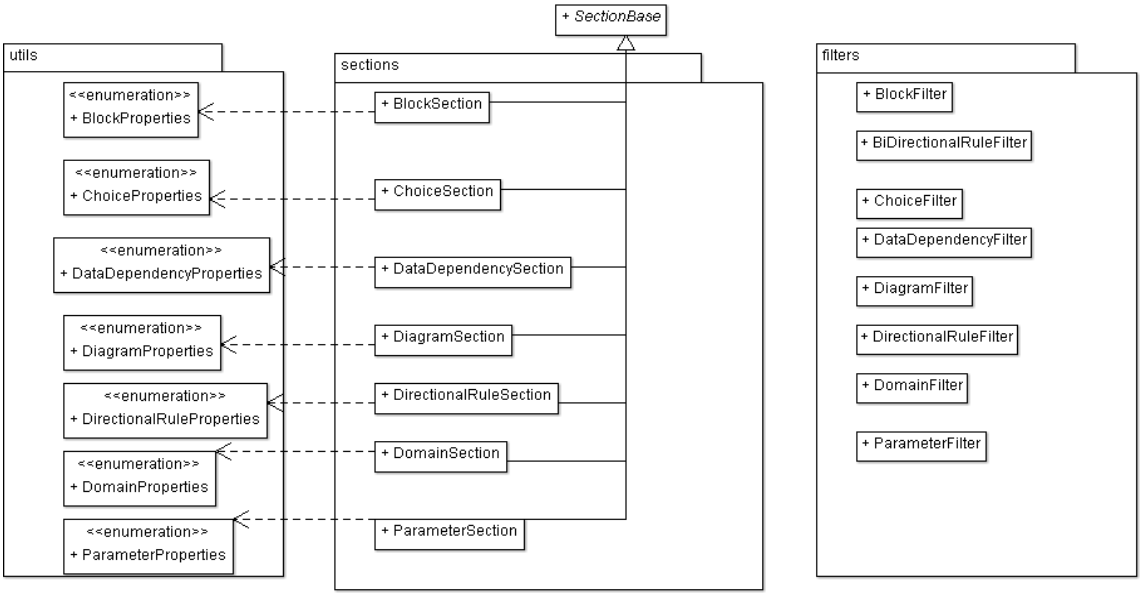


Imagen 17 : Paquetes - Diagrama clases Properties

5.2.2 Clases base

Han sido implementadas una serie de clases base a fin de servir de punto de extensión común para funcionalidades que comparten características y evitar duplicidad de código.

A continuación, veremos el objetivo que cubre cada una de esas clases base, agrupadas por característica que manejan:

Transformation [Paquete adds.base]

Se define una clase base que maneje una instancia única para trabajar con el objeto Transformation del diagrama XMI. La primera vez que se trabaja con el diagrama, se comprueba si existe ya en el diagrama creándolo si no existe y añadiéndolo al diagrama.

- **Creates [Paquete creates.base]**
  - Se define clase base para la creación de dominios CreateDomainBaseFeature, al existir domain_to y domain_from donde tan sólo cambia el objeto de dominio que se crea.

    Las subclases deberán implementar el método create para definir el objeto concreto de dominio en cada caso.

  - Se define clase base para la creación de relaciones CreateBaseReferenceFeature, objetos de dominio que extienden de Flow, a fin de reunir las condiciones comunes de las relaciones. Se controla la posibilidad de creación, y de origen de la creación, de la relación al coincidir en las relaciones los elementos origen y destino posibles (Block y Rules). Además, se recoge la posibilidad de controlar la duplicidad, si existe ya una relación que se desea crear, que es requerida para ciertas relaciones.

    Las subclases deberán implementar el método createObjectFeature para definir el objeto concreto de dominio en cada caso.

- **Adds [Paquete adds.base]**
  - Se define la clase base AddBaseFeature para el manejo centralizado, de la instancia Transformation, y ofrecer funcionalidades para añadir cada figura de manera adecuada al fichero XMI.

Además ofrece métodos de uso común en las funcionalidades de añadir como pueden ser la obtención del diagrama del contenedor, la obtención del contenedor sobre el que se encuentra añadida una figura, etc.

Esta clase la extenderán el resto de clases base de este grupo.

Las subclases deberán implementar el método add para configurar el diseño de la figura en el diagrama y el método canAdd para permitir o no añadir dicha figura al diagrama según corresponda.

- Se define clase base AddBaseRuleFeature para la creación de figuras que extienden de Rule, DirectionalRule y BiDirectionalRule. En esta clase se da valor a los atributos del objeto de dominio y se crea la representación completa de los objetos al ser idéntica.

  Las subclases deberán implementar el método canAdd para permitir o no añadir dicha figura al diagrama según corresponda, ya que en este caso las reglas deben comprobar si el diagrama permite el uso de ese tipo de reglas.

- Se define la clase base AddBaseFlowFeature para la creación de conexiones de formato estándar. Como estándar se define un diseño común, flecha discontinua y se da la posibilidad de elegir si contiene un círculo en su posición central. Lo que varía y debe extender cada clase superior es el nombre a mostrar sobre la relación. Se ha utilizado para la relación After pero deja abierto su uso a futuras relaciones si fuera preciso.

  Las subclases tan sólo deben implementar el método getNameFlow para indicar el texto sobre la flecha e indicar en el constructor si debe mostrar círculo o no en su imagen.

- **Layout [Paquete layouts.base]**

  - Se define la clase LayoutBase para el control base de layout de figuras aportando el control de comprobación de tamaño mínimo por defecto que debe tener cada figura.

    Las subclases deben implementar los métodos canLayout y layout para indicar respectivamente si se puede manejar el layout y controlar los redimensionamientos.

- **Section [Paquete properties.sections.base]**

  - Se define la clase como punto de extensión para todas las clases Section. No se aporta en la actualidad funcionalidad base, pero sirve para marcar las extensiones e implementaciones que requiere una clase de este tipo.

    Las subclases en este caso deben realizar toda la implementación completa, diseño de propiedades a mostrar, manejo de cambios y actualización del objeto de dominio.

6 Implementación

En el punto anterior hemos visto de manera global la arquitectura y features que se manejan en el editor a desarrollar, que marcan el desarrollo y la utilidad de las clases base que facilitan la implementación.

Veremos a continuación, en primer lugar el IDE de desarrollo que se ha precisado y después la implementación concreta que ha requerido cada feature [Sección 5.1.2] en relación a los elementos del lenguaje de reglas [Sección 3] que maneja el editor.

6.1 Entorno de Desarrollo

Para la implementación se ha creado un proyecto de tipo plug-in para Eclipse. El proyecto se apoya en un conjunto de proyectos que conforman el lenguaje de reglas de transformación en que se basa el proyecto transML. Como ya se ha comentado, se hace uso de ciertos plug-in de Eclipse para trabajar en el proyecto, cabe citarlos y ver la lista de extensiones requeridas para el desarrollo del proyecto:

Plug-in instalados en Eclipse, por orden de incorporación al desarrollo del proyecto.

- Graphiti 0.9.1
- Epsilon 1.0 + Eugenia 1.0

Extensiones de Plug-ins utilizados en el proyecto [Imagen 18]:

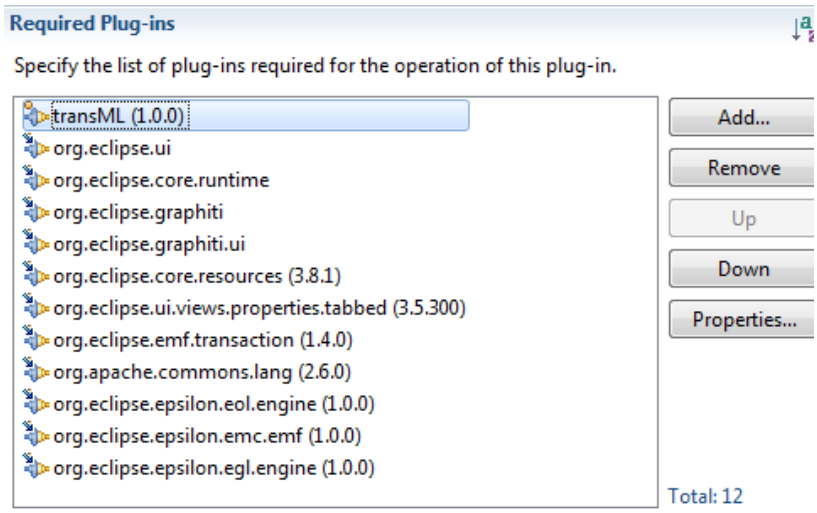


Imagen 18 : Implementación - plug-ins

6.2 Create

Cada elemento ha precisado una clase independiente para permitir crear la figura en el editor.

Para las figuras, que no son relaciones, se extiende la clase AbstractCreateFeature y se sobrescribe el método que comprueba si es posible crear cada componente sobre el diagrama y el método que crea el objeto de dominio relacionado, cambiando según el objeto de dominio las condiciones y objeto manejado.

Figuras Domain y relaciones manejan clase base como ya se ha comentado.

6.3 Add

Cada elemento ha precisado una clase independiente para permitir añadir, finalmente, la figura en el editor. En este punto, las figuras requieren un desarrollo independiente del pictograma.

Todas las figuras deben registrar su objeto de dominio en el fichero XMI de manera adecuada.

Para las figuras que no son relaciones, se extiende la clase base AddBaseFeature. Para las figuras DirectionalRule y BiDirectionalRule, se extiende la clase base AddBaseRuleFeature. Partiendo de esta clase base tan sólo es preciso extender el método de posibilidad de creación ya que se debe controlar la configuración del diagrama, si permite reglas direccionales o bi-direccionales.
Para la figura After se extiende la clase base AddBaseFlowFeature al ser una relación considerada básica.

Ciertas figuras han requerido **implementaciones especiales** para aportar las características que requerían. Veamos en detalle cada una de ellas:

- Para las figuras Domain, se ha requerido una implementación combinada. Se parte de la creación del pictograma, y se crea además una relación, al mismo tiempo, entre el dominio y la regla sobre el que se crea. Con esto se aportará movimiento a la figura en torno a su regla.

- Para la figura Parameter, se ha requerido una implementación combinada. Se parte de la creación de una figura estándar, con su representación rectangular y se crea además una relación, al mismo tiempo, entre el parámetro y el dominio sobre el que se crea. Con esto se aportará movimiento a la figura en torno a su dominio.

- Para las figuras Call y When, se ha requerido una implementación donde se diseña la figura circular central y se crean conexiones de origen a círculo y desde el círculo al destino. Esto es necesario para poder relacionar posteriormente la figura central con parámetros para crear la relación DataDependency.

- Para la figura Choice, se ha requerido una implementación más compleja que las anteriores. Por un lado para una primera conexión Choice se crea el pictograma rombo central de la conexión, conexión origen a rombo y de rombo a destino.

  A partir de aquí, sobre una figura Choice se pueden añadir entradas o salidas. Si se relaciona un componente con el rombo del Choice se crea una relación del lado source, y si se relaciona el rombo del Choice con un destino se crea una relación en la parte target.

6.4 Update

Por defecto se utiliza la implementación de Graphiti para el manejo de esta característica.

Para ciertas figuras se ha requerido una extensión concreta a fin de realizar las modificaciones de la figura de manera oportuna:

- Se ha creado un manejador de updates común para las figuras Rules, Domain, Parameter y DataDependency para manejar el cambio del texto de su nombre.

- Las figuras DirectionalRule y BiDirectionalRule han precisado un Update específico ya que el cambio de propiedades en estas figuras propicia cambios en la representación de su pictograma. Así además del cambio de nombre se comprueban cambios en otras propiedades propias de figuras Rule.

6.5 Delete

Por defecto se utiliza la implementación de Graphiti para el manejo de esta característica.

Para la mayoría de las figuras se ha requerido una extensión concreta a fin de realizar un borrado controlado tanto de representaciones en el diagrama como a nivel del fichero XMI eliminando los elementos correctamente del objeto Transformation.

Veamos en detalle las eliminaciones especiales de las figuras que lo requieren:

- Figura Block necesita control de borrado ya que se deben eliminar en cascada los elementos relacionados y relaciones de las que forma parte él o sus componentes internos.

- Figuras DirectionalRule y BiDirectionalRule necesitan control de borrado ya que se debe eliminar en cascada los elementos relacionados, dominios y parámetros, y las relaciones de las que forma parte.

- Figura Domain necesita control de borrado para eliminar en cascada los parámetros relacionados y relaciones de las que forma parte.

- Para el resto de figuras se ha creado una clase común donde se controla la prohibición de borrado de elementos intermedios de conexión que no son elementos de dominio.

- Para las relaciones Choice especiales origen-rombo y rombo-destino se eliminan tan sólo la referencia en source o target dentro del elemento Choice.

  En caso de no existir más entradas o salidas se avisa al usuario de la no posibilidad de realizar la acción, teniendo que eliminar manualmente la figura Choice, el rombo, si se desea eliminar la figura Choice completa.

6.6 Move

Por defecto se utiliza la implementación de Graphiti para el manejo de esta característica.

Para ciertas figuras se ha requerido un control más en detalle de sus movimientos. Veamos en detalle los controles creados:

- Figura Block, se controla que se mueva sólo sobre el diagrama o sobre el Block al que pertenece.

- Figuras Rule, se controla que se mueva sólo sobre el diagrama o sobre el Block al que pertenece. Además se controla movimiento no muy lejos de sus relaciones con dominios a fin de que permanezcan lo más unidos posibles.

- Figura Domain, se controla que no se mueva sobre otras figuras y que su movimiento se mantenga dentro de un rectángulo imaginario alrededor de la figura Rule de la que parte.

- Figura Parameter, se controla que no se mueva sobre otras figuras y que su movimiento se mantenga dentro de un rectángulo imaginario alrededor de la figura Domain de la que parte.

6.7 Layout

Tan sólo se permite cambios de tamaño de la figura Block, la cual debe mantener un tamaño mínimo por defecto. Para el resto de figuras se ha bloqueado el cambio de tamaño para simplificar su manejo.

6.8 Properties

Para el manejo de las propiedades, cada figura ha requerido la implementación de una clase, conocida como Section, donde se diseña la hoja de propiedades con los campos adecuados en relación a los atributos modificables en cada caso.

Además, se requiere otra clase por cada figura, conocida como Filter que indica la relación del objeto de dominio y la hoja de propiedades que lo maneja.

Por otro lado, es preciso declarar en el fichero plugin.xml, de configuración del proyecto, una extensión que indique que para un Filter concreto se debe mostrar una hoja de propiedades (Section) concreta.

Veamos un ejemplo de extensión incluida en el plug-in para conseguirlo:

```
<extension point="org.eclipse.ui.views.properties.tabbed.propertyTabs">
        <propertyTabs contributorId="Proyecto.PropertyContributor">
         <propertyTab id="graphiti.main.tab.Block" label="Block"/>
        </propertyTabs>
</extension>

<extension point="org.eclipse.ui.views.properties.tabbed.propertySections">
 <propertySections contributorId="Proyecto.PropertyContributor">
  <propertySection
     class="proyecto.features.properties.sections.BlockSection"
     filter="proyecto.features.properties.filters.BlockFilter"
     id="graphiti.main.tab.BlockSection"
     tab="graphiti.main.tab.Block">
  </propertySection>
 </propertySections>
</extension>
```

6.9 Generación de código

Para la generación de código se ha precisado implementar una clase que extienda AbstractCustomFeature, lo que permite, a través del menú contextual del diagrama, proceder a la generación de código del diagrama actual.

El proceso trabaja con el modelo actual del diagrama, Transformation [Sección 3.1] con objetos de dominio. Se han implementado unas plantillas de EGL [12] en las que se trabaja con el modelo para obtener la salida de código generado en un fichero.

Veamos una muestra del estilo que siguen estas plantillas, aparece marcado la parte que se volcaría en el fichero de código generado de salida:

```
for (rule in transformation.rulesFlow()) {
   if (rule.rationale.isDefined()=true and rule.rationale.trim().length()>0) {%]
// [%=rule.rationale%]
[% }
   if (rule.isAbstract.isDefined()=true and rule.isAbstract) {%]
@abstract
[% }
   if (rule.isLazy.isDefined()=true and rule.isLazy) {%]
@lazy
[% }
   if (rule.domain_from.isDefined() and rule.domain_from.size()>0 and
   rule.domain_to.isDefined() and rule.domain_to.size()>0) {
      var domain_from := rule.domain_from.first();
      var param_from  := domain_from.params.first%]
rule [%=rule.name%]
    transform [%=param_from.name%] :
[%=domain_from.name%]![%=param_from.type.name%]
    to        [%
```

Como podemos ver, en la plantilla se trabaja con el modelo de dominio usando la raíz transformation y leyendo la configuración de los objetos del diagrama, el valor de sus atributos marcará la configuración de la salida, incluyendo las líneas de salida si corresponde a los datos leídos.

La salida se guarda en un fichero que puede ser ejecutado usando el motor de ejecución del lenguaje de transformación destino seleccionado, que en nuestro caso es ETL.

7 Resultados

En este punto veremos, mediante la explicación de capturas de pruebas realizadas manejando el plug-in implementado, el resultado final del proyecto.

En primer lugar, veremos las representaciones de las figuras diseñadas y su comportamiento dentro del editor gráfico.

Por otro lado, se mostrarán ejemplos de salidas de código generado por medio de diagramas diseñados con el plug-in.

7.1 Editor gráfico

Al crear un diagrama con el editor, tenemos un diagrama en blanco con una paleta lateral [Imagen 19] con todos los elementos que podemos añadir al diagrama.

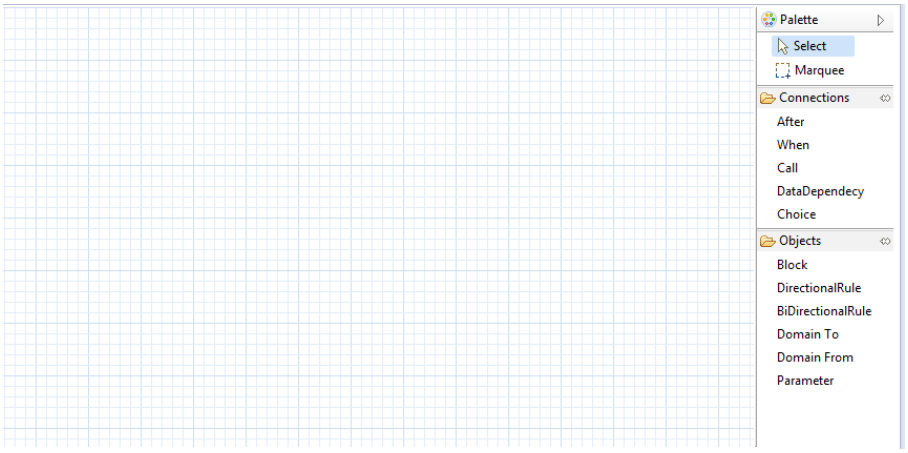


Imagen 19 : Resultados - Editor y barra herramientas

Veamos a continuación la figura obtenida en cada caso en el editor, junto con sus hojas de propiedades.

7.1.1 Diagrama

A nivel de diagrama no hay diseño como tal, pero sí se ha creado una hoja de propiedades que aparece al posicionarse sobre un diagrama vacío [Imagen 20], donde se puede seleccionar si se trabajará con reglas uni-bidirecionales o bi-directionales.

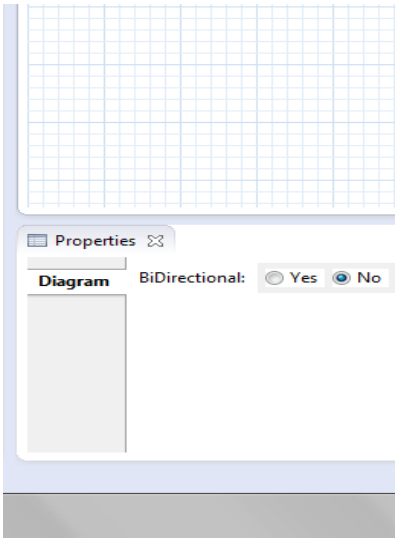


Imagen 20 : Resultados - Propiedades a nivel de diagrama

Una vez que se añaden figuras al diagrama no se puede cambiar su configuración y al intentar agregar una figura BiDirectional [Imagen 21], en este caso al estar marcado el diagrama con la propiedad a “No”, avisa de la situación y no permite añadir la figura.

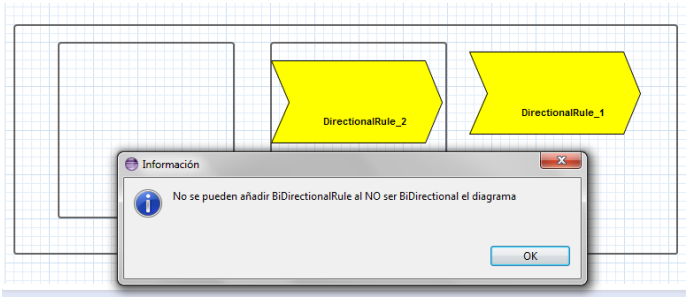


Imagen 21 : Resultados - Diagrama No BiDirectional

7.1.2 Block

Como vemos en la ilustración [Imagen 22], la figura Block permite incluir figuras en su interior y maneja una hoja de propiedades según las especificaciones.

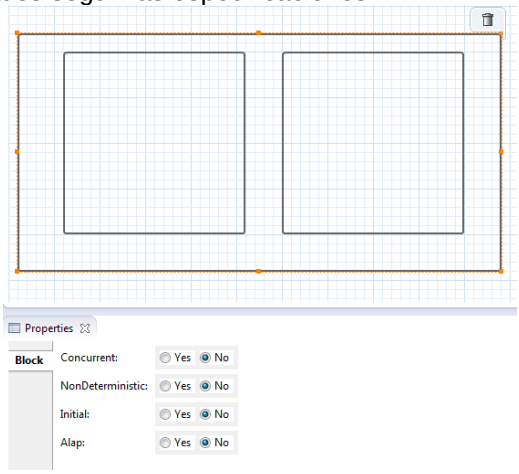


Imagen 22 : Resultados - Propiedades Block

7.1.3 DirectionalRule y BiDirectionalRule

Las siguientes capturas son igualmente válidas para BiDirectionalRule al compartir exactamente su diseño y lógica.
En esta imagen [Imagen 23], podemos ver una figura DirectionalRule con sus componentes anidados Domain y Parameter sobre estos.

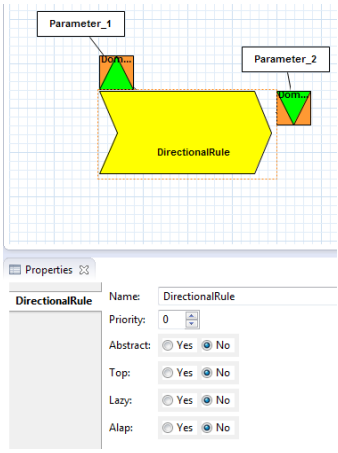


Imagen 23 : Resultados - Figura DirectionalRule

Según los valores de atributos indicados en la hoja de propiedades la figura puede cambiar su diseño dinámicamente. Así al cambiar la propiedad Priority a un número mayor que cero vemos que aparece dicho dato en la esquina superior derecha [Imagen 24].

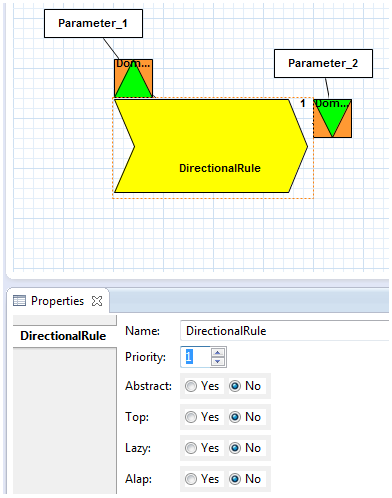


Imagen 24 : Resultados - Diagrama Rule Priority > 0

Si se modifica el atributo Top a un valor “Yes” aparece una doble línea interna en el diseño [Imagen 25].

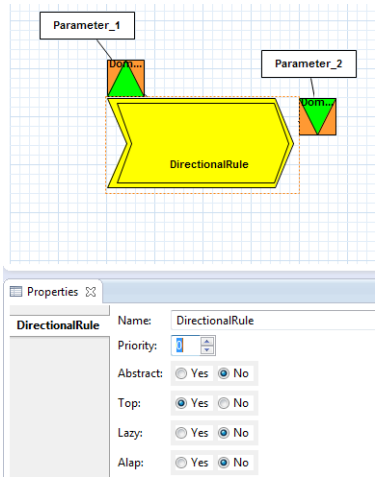


Imagen 25 : Resultados - Diagrama Rule Top "Yes"

7.1.4 Domain

La figura Domain puede tomar dos representaciones en el diagrama, según la dirección del mismo (To o From). En la siguiente imagen [Imagen 26] vemos ambos ejemplos junto con la hoja de propiedades asociada al elemento To.

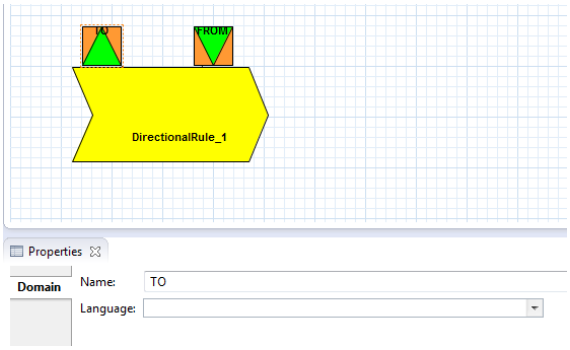


Imagen 26 : Resultados - Figuras DomainTo y DomainFrom

Es de destacar el movimiento que puede realizar una figura Domain alrededor de su figura "padre" [Imagen 27]. Como vemos partiendo de la posición inicial de creación es posible disponer múltiples Domain en torno a una regla.

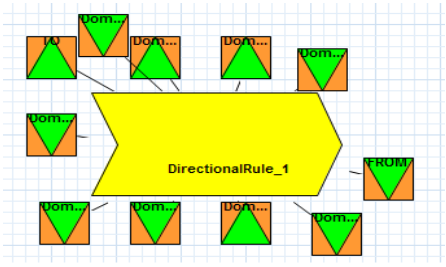


Imagen 27 : Resultados - Diagrama múltiples Domain en Rule

7.1.5 Parameter

La figura Parameter se debe añadir sobre una figura Domain. Esta figura [Imagen 28], puede moverse alrededor de su "padre" Domain dentro de unos límites marcados y se pueden añadir tantos Parameter como se deseen, dos en el ejemplo.

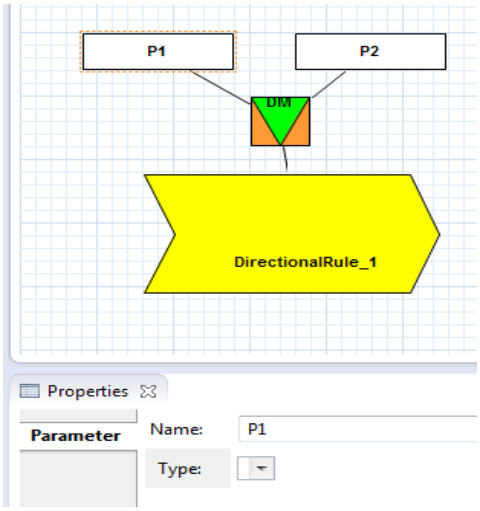


Imagen 28 : Resultados - Diagrama Parameter

Para tomar valor de Type en sus propiedades, la figura Domain debe haber tomado valor en su propiedad Language para poder seleccionar en Parameter un Type [Imagen 29]. En caso de cambiar el Language en Domain, el Type elegido en Parameter se elimina volviendo a la primera parte de la imagen.

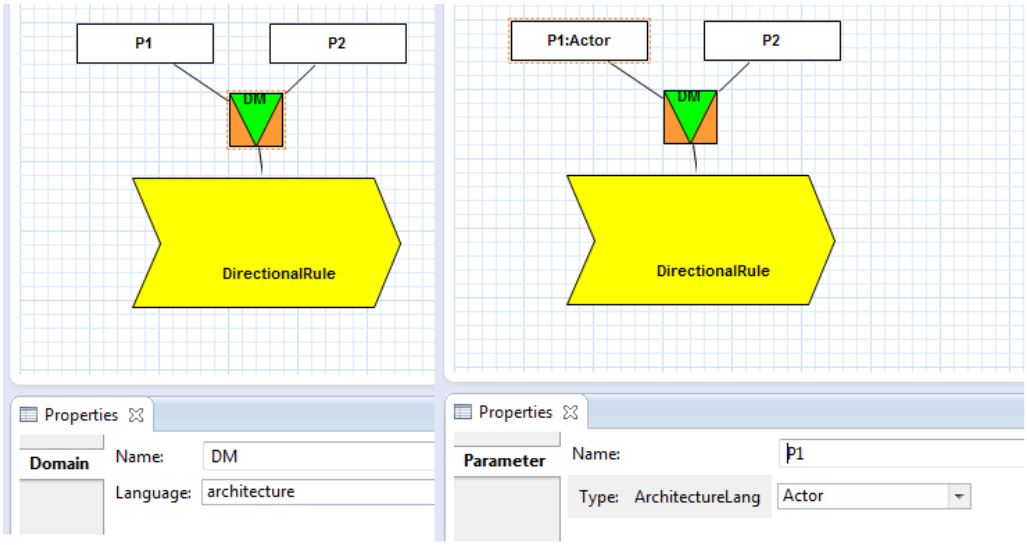


Imagen 29 : Resultados - Asignación propiedades Domain-Parameter

7.1.6 After

La relación After [Imagen 30], se puede producir entre bloques y reglas, pudiendo ser el mismo origen-destino.

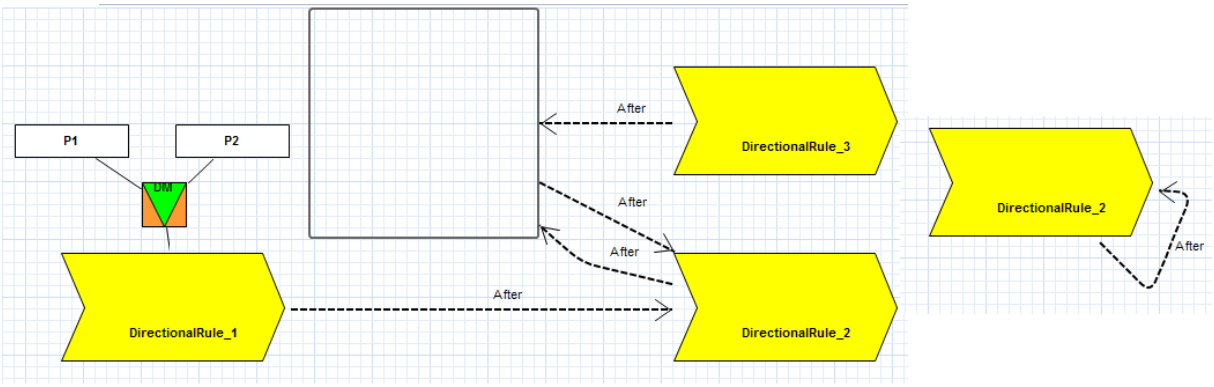


Imagen 30 : Resultados - Relación After

Como restricción, no puede realizarse la misma conexión [Imagen 31] dos veces entre los mismos origen-destino.

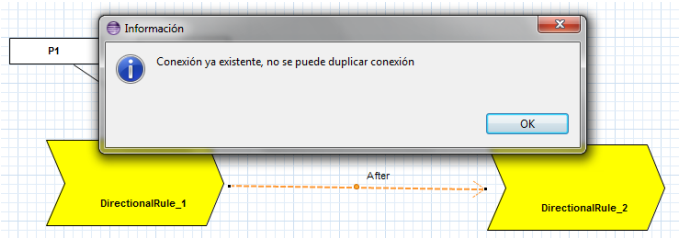


Imagen 31 : Resultados - Restricción uso After

7.1.7 When

Al igual que la relación After, When [Imagen 32] puede relacionar bloques, reglas y consigo mismo.

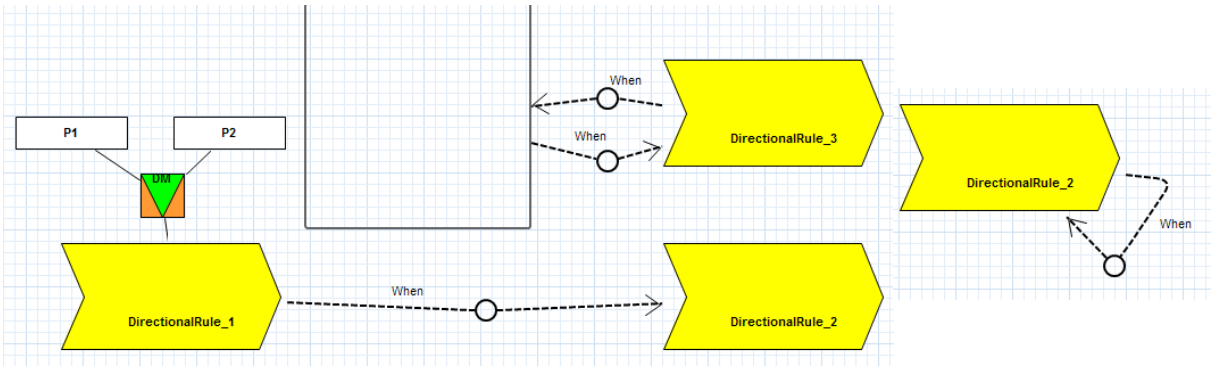


Imagen 32 : Resultados - Relación When

Igualmente no se pueden realizar conexiones When duplicadas [Imagen 33].

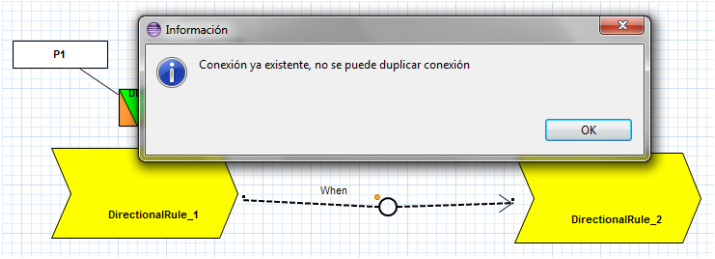


Imagen 33 : Resultados - Restricción uso When

7.1.8 Call

La relación Call [Imagen 34] permite igualmente relaciones entre bloques y reglas.

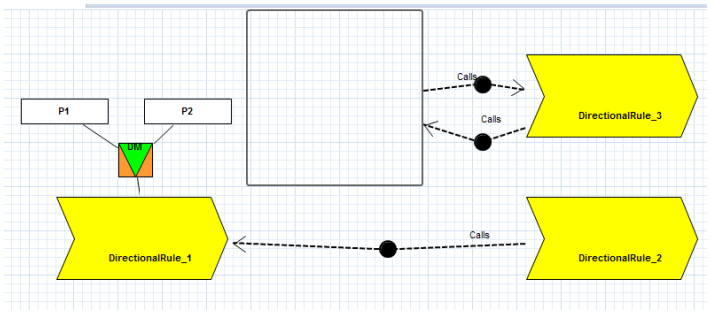


Imagen 34 : Resultados - Relación Call

En este caso no existe la restricción de duplicidad de relaciones origen-destino [Imagen 35].

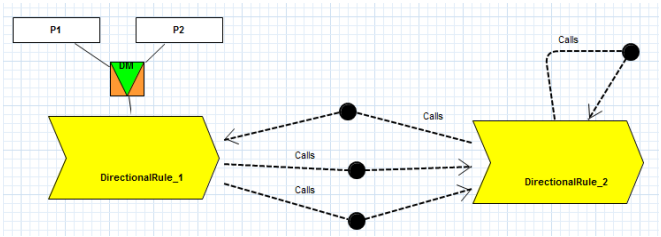


Imagen 35 : Resultados - No restricción Call

7.1.9 DataDependency

La relación DataDependency puede tener como origen When o Call, veremos los ejemplos con When. Como única restricción [Imagen 36], es preciso crear la relación con destino un parámetro de la regla destino apuntada por la relación Call o When.

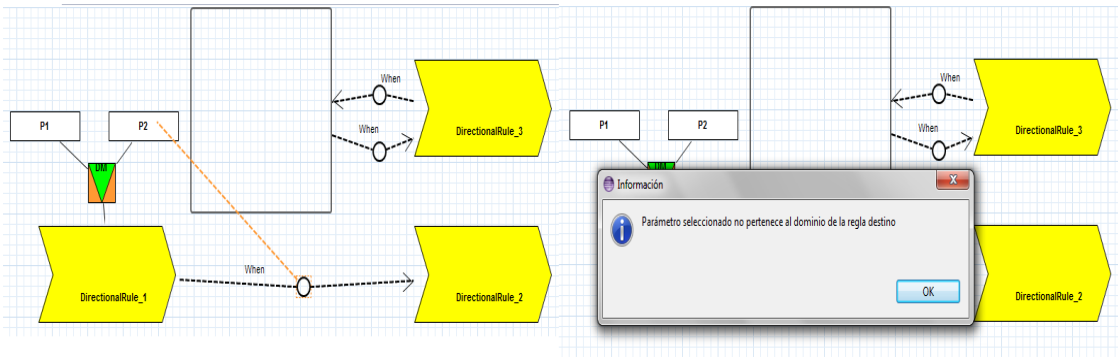


Imagen 36 : Resultados - Relación DataDependency

La relación se crea correctamente eligiendo un destino adecuado, parámetro de regla destino. En este momento ya se puede elegir el parámetro origen [Imagen 37].

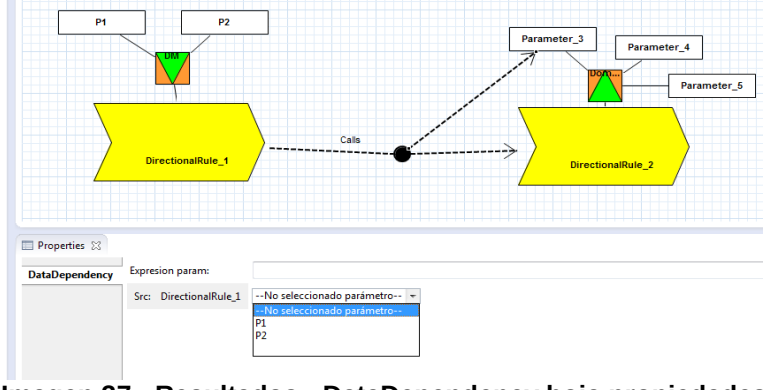


Imagen 37 : Resultados - DataDependency hoja propiedades

Creada la relación se puede indicar Expresion param o indicar un parámetro de la regla origen [Imagen 38].

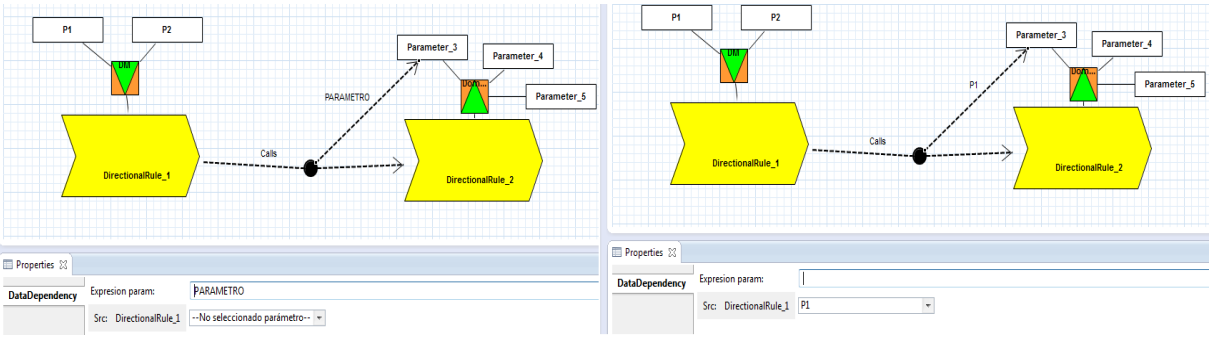


Imagen 38 : Resultados - DataDependency asignar parámetro

7.1.10 Choice

La relación Choice se puede crear entre bloques y reglas, pudiendo realizarse sobre la propia regla origen-destino [Imagen 39]. En su hoja de propiedades tan sólo se define la restricción adecuada [Imagen 40].

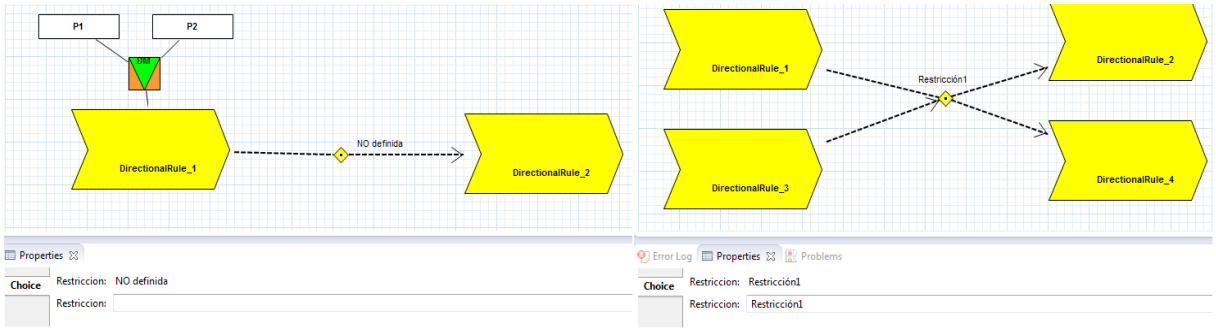


Imagen 39 : Resultados - Relación Choice y restricción

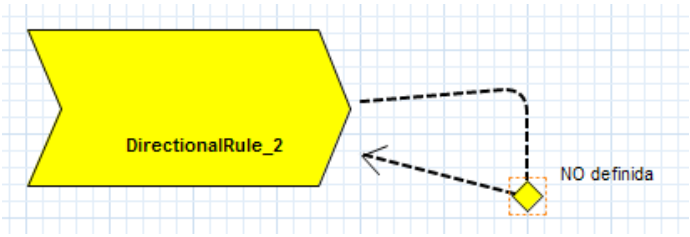


Imagen 40 : Resultados - Choice sobre misma regla

7.1.11 Validaciones generación código ETL

Previo a la generación de código ETL, se realizan unas series de validaciones para comprobar la validez del diagrama diseñado, dentro los límites de uso de ETL.

Un diagrama vacío no tiene sentido [Imagen 41], por lo que intentar generar código devolverá error.

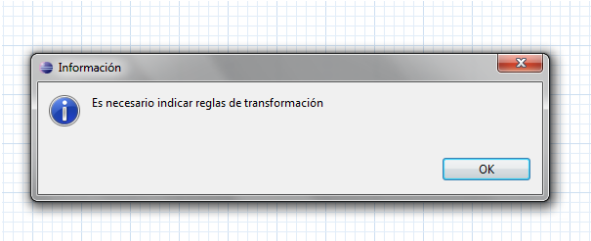


Imagen 41 : Resultados - Validación Generación diagrama vacío

Así por ejemplo se comprueba que una regla tiene exactamente un dominio de entrada y uno de salida [Imagen 42], mostrando error si no se cumple.

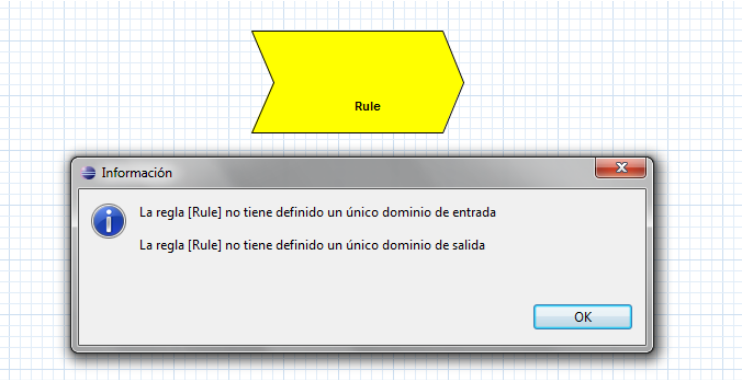


Imagen 42 : Resultados - Validación Generación sin dominios

Pasada esta validación, se comprueba que cada dominio tiene definida su propiedad language [Imagen 43] y un parámetro asignado [Imagen 44]. Cada parámetro deberá tener además su propiedad type seleccionada.

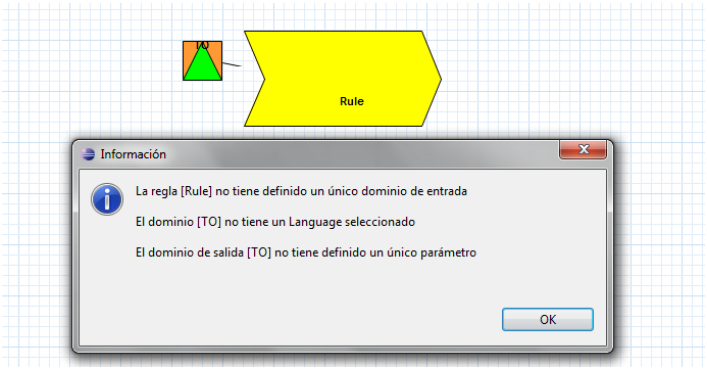


Imagen 43 : Resultados - Validación Generación sin languages y/o parámetros

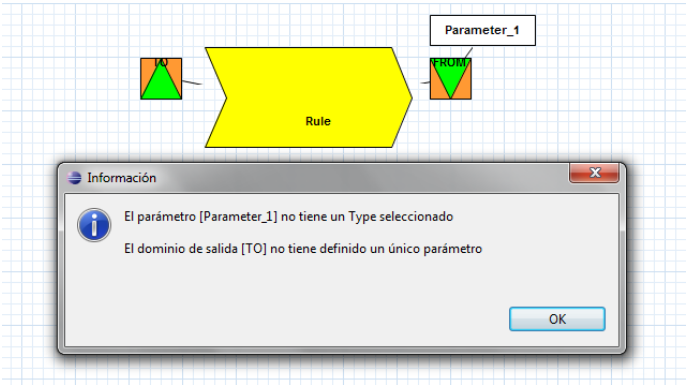


Imagen 44 : Resultados - Validación Generación sin parámetros

7.2 Ejemplo completo

En este punto se mostrará por medio de capturas, el proceso completo de transformación desde un modelo origen, conocido y que llamaremos OO.model, a un modelo destino, que deseamos generar por medio del aplicativo y llamaremos DB.model.

En primer lugar, partimos de un diagrama en blanco [Imagen 45], y se indica la propiedad del diagrama de trabajar con reglas unidireccionales.

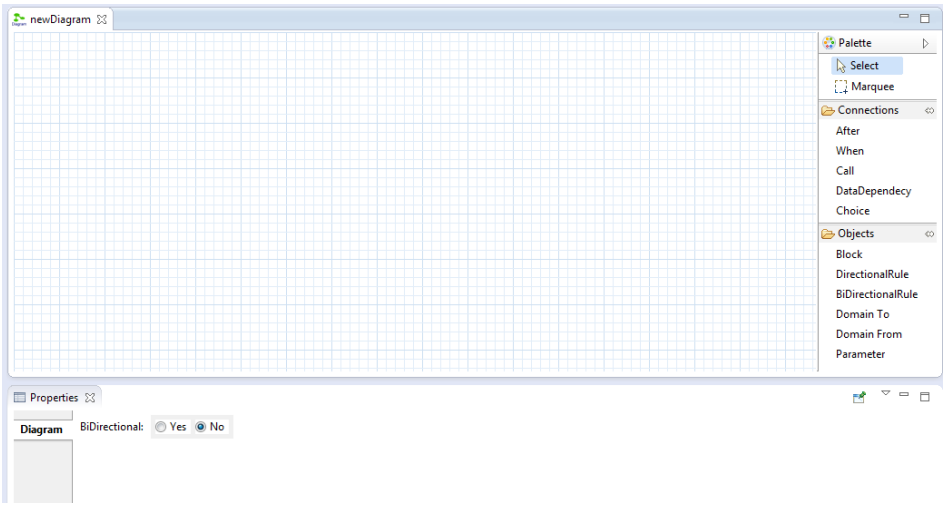


Imagen 45 : Resultados [Ejemplo] - Paso 1: Diagrama en blanco

A continuación, se añade una regla [Imagen 46], con los valores por defecto de atributos.

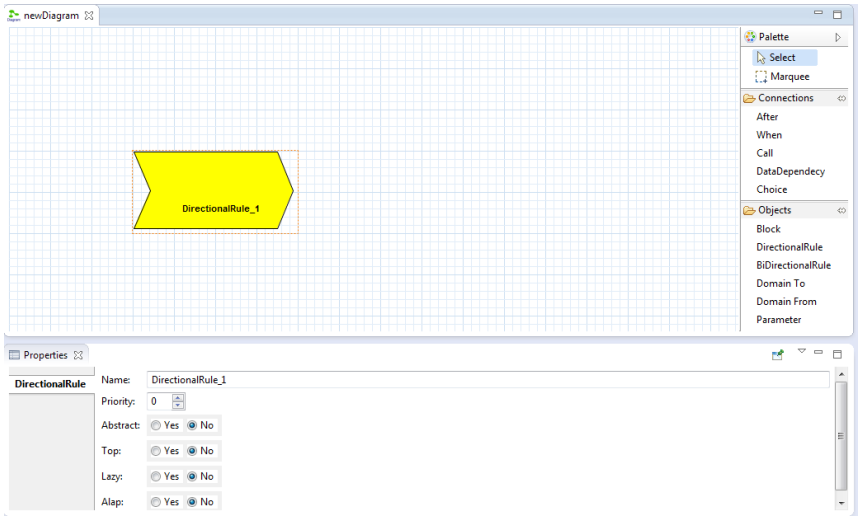


Imagen 46 : Resultados [Ejemplo] - Paso 2: Añadir regla transformación

Se modifica el nombre de la regla para mayor aclaración del objetivo de la regla. En este caso la llamaremos “Class2Table”, al ser como veremos en pasos siguientes, “Class” el tipo del modelo origen y “Table” el tipo del modelo destino.

Además se añade un dominio de entrada, de nombre OO y bajo el Language models.OO, que como ya hemos comentado al inicio, es nuestro modelo origen. A su vez, se añade un dominio de salida, de nombre DB, y bajo el Language models.DB que es modelo destino del caso mostrado [Imagen 47].

En la parte derecha de la imagen [Imagen 47], vemos cómo se puede elegir el Language (o metamodelo) y que aparecen todos los metamodelos registrados en Eclipse. Para la prueba se han registrado previamente los metamodelos del ejemplo en Eclipse para poder seleccionarlos.

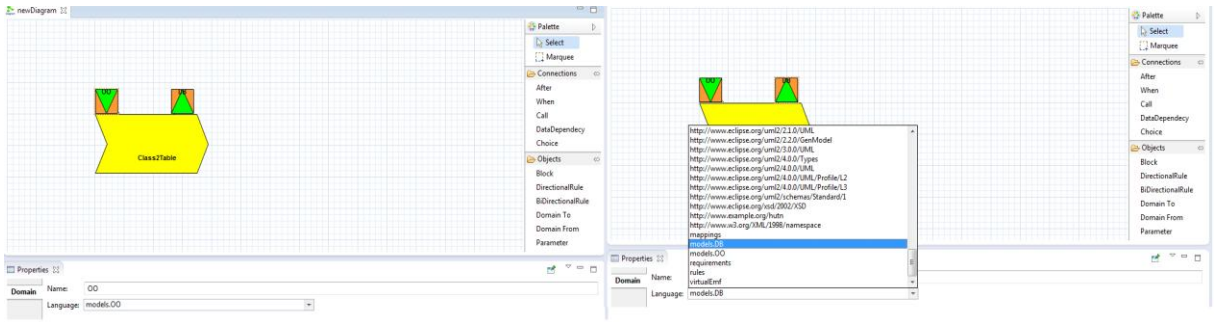


Imagen 47 : Resultados [Ejemplo] - Paso 3: Añadir Dominios

Configurados los dominios, es momento de añadir los parámetros [Imagen 48] que indicarán los tipos de datos que participan en la transformación.

Se configura el dominio de entrada bajo el tipo “Class” y el dominio de salida bajo el tipo “Table”.

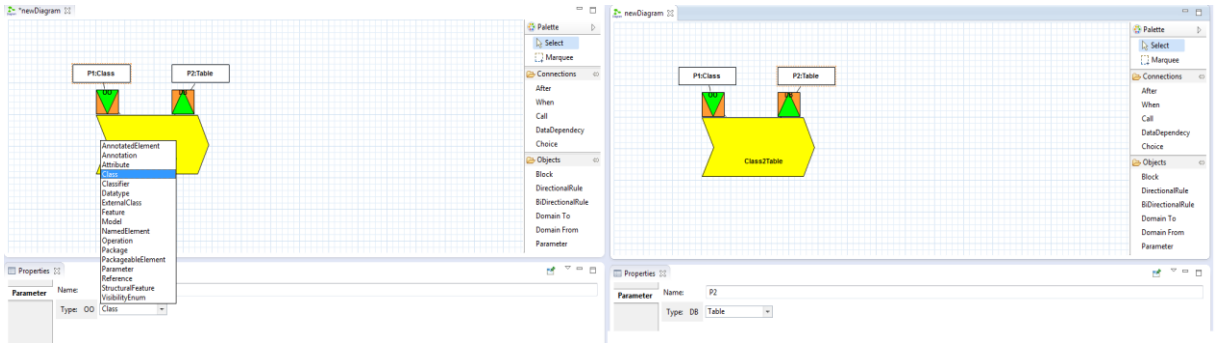


Imagen 48 : Resultados [Ejemplo] - Paso 4: Añadir Parámetros

Terminado el diseño de la transformación se procede a generar el código para el lenguaje implementado [Imagen 49], en este caso ETL que es el implementado en el proyecto. Se indica ruta de salida donde el aplicativo dejará el fichero de salida de nombre tr_transformation.etl.

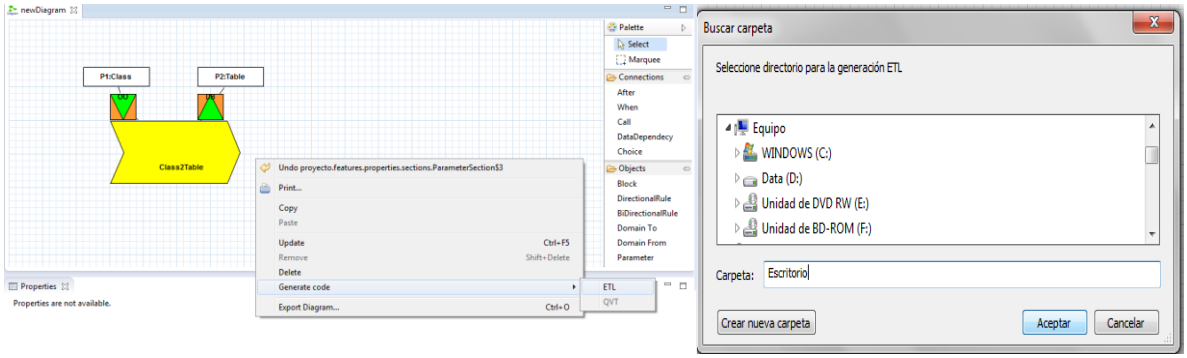


Imagen 49 : Resultados [Ejemplo] - Paso 5: Generación código

Se obtiene un fichero de salida [Imagen 50] en lenguaje adecuado para realizar la transformación de modelos.

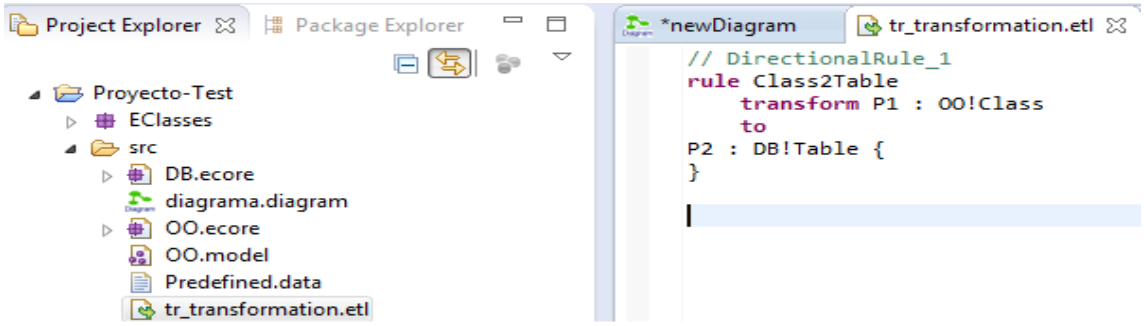


Imagen 50 : Resultados [Ejemplo] - Paso 6: Salida fichero ETL

Finalmente, a modo de comprobación se ha probado a ejecutar el fichero de salida obtenido con el aplicativo, para verificar su corrección.

Siguiendo los siguientes pasos [Imagen 51] para ejecutar el fichero de transformación, configurando los datos de modelos de entrada y salida, se obtiene el modelo destino que se buscaba con la transformación, nombrado como DB.model.

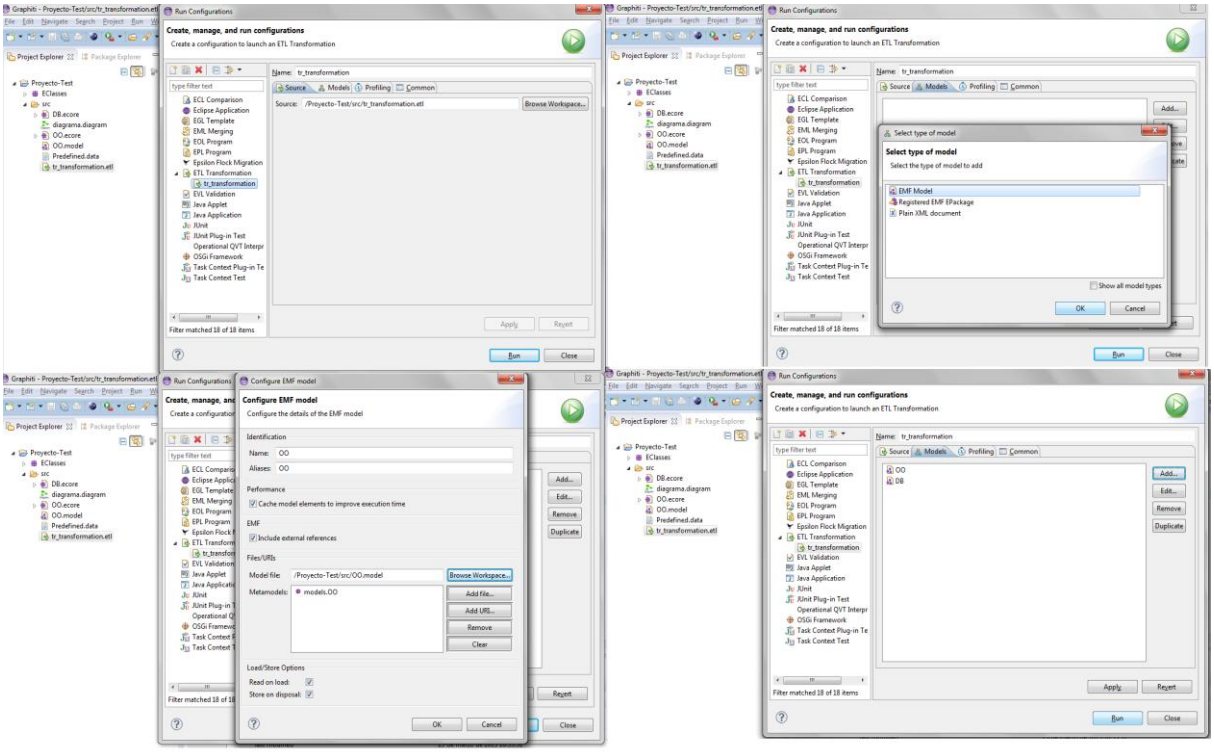


Imagen 51 : Resultados [Ejemplo] - Paso 7: Configuración Run Fichero ETL

Tras ejecutarlo obtenemos el Modelo destino de la transformación que se pretendía [Imagen 52]. Se obtiene el modelo que se buscaba con la transformación, nombrado como DB.model.

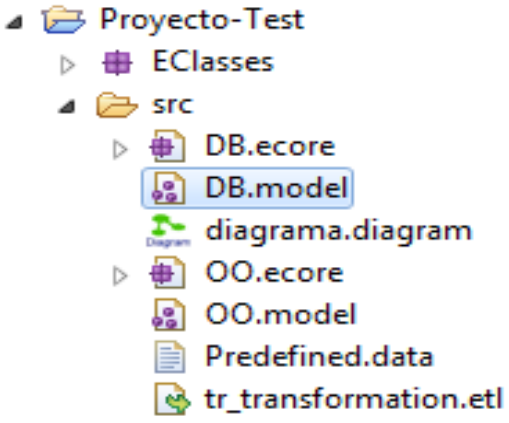


Imagen 52 : Resultados [Ejemplo] - Paso 8: Se obtiene modelo destino

7.3 Plug-in

Como punto final del proyecto, se creó un proyecto de tipo plug-in en eclipse para generar el plug-in del trabajo de fin de grado desarrollado. La creación del plug-in es directa, tan sólo es necesario indicar la relación de proyectos involucrados, en nuestro caso el proyecto implementado y generar la creación del plugin. Para la prueba se eligió salida de plugin en fichero.

Por último se abrió una instancia de Eclipse y por medio de su menú de instalación de nuevo software se seleccionó el archivo creado, instalando con éxito el plug-in del proyecto en Eclipse, pudiendo trabajar perfectamente con todas las partes de la funcionalidad desarrollada.

8 Conclusiones y trabajo futuro

En esta sección se expondrán las conclusiones obtenidas tras la realización del proyecto y se comentarán los puntos de extensión o mejora en futuras continuaciones del trabajo.

8.1 Conclusiones

La realización de este proyecto ha permitido aportar una definición a alto nivel para la creación de diagramas de transformación, basados en el lenguaje de reglas de transformación de motivación del proyecto.

Además se incluye funcionalidad, parcial por el alcance del proyecto, para la generación de código a partir de los diagramas diseñados que demuestre la utilidad final y tangible de la línea de investigación inicial, que no es otra que la transformación completa desde un modelo a código.

Se ha logrado diseñar cada elemento del lenguaje bajo los requisitos ideados de representación y principios básicos de comportamiento, consiguiendo un diseño intuitivo y fluido de diagramas de manera gráfica. Si bien aunque el diseño sea intuitivo, se requiere del conocimiento de las reglas que se manejan para dar sentido e interpretación al diagrama diseñado.

Se ha conseguido además, bajo la elección realizada de Frameworks, Eclipse y Graphiti, sobre los que se ha construido el desarrollo, crear un plug-in de la funcionalidad. Una herramienta completa que se puede integrar en la plataforma Eclipse de cualquier desarrollador para trabajar con ella.

Como punto débil, aunque conocido y no tratado en el alcance del proyecto, en el actual proyecto se han definido reglas de diseño según el modelo seguido pero no se ha definido un punto de validación de la transformación que se diseña, recayendo este punto en la experiencia del desarrollador en esta área.

En resumen, se ha construido de manera completa el editor gráfico planteado en diseño y se ha demostrado que cumple su cometido principal, de representación y de correcta interpretación para la generación final de código. Además de sentar una base de diseño y funcionalidad extensible en futuros trabajos.

8.2 Trabajo futuro

Como se comentó en la introducción del proyecto, el objetivo principal era aportar una herramienta para la edición gráfica de los diagramas de reglas de transformación, área para la que no existía dicha funcionalidad.

Según esto, aunque el proyecto cubra los principios básicos definidos, al ser una primera aproximación real, se considera que puede ser ampliable y mejorable en varios de sus conceptos de cara a futuros trabajos de continuación.

Respecto las líneas de trabajo futuro, de continuación de este proyecto se pueden destacar:

- Ampliar y mejorar los aspectos de layout de las figuras, movimientos y diseño de los elementos gráficos.

- Añadir funcionalidad de creación del diseño guiada por menús contextuales, en figuras que pueden añadir elementos, en lugar de arrastrar todos los componentes sobre el diagrama o sobre una figura, a fin de facilitar la creación rápida de diagramas.

- Siguiendo la línea del punto anterior, aportar control de validación del diagrama en construcción a fin de guiar al diseñador en la falta de lógica de cara a la interpretación final para la generación de código.

- Ampliar el diseño de plantillas EGL, o de otro lenguaje si se desea, con la interpretación completa de todos los elementos del diagrama para poder generar código de diagramas complejos.

9 Bibliografía

[1] Wikipedia, MDE. Available at : http://en.wikipedia.org/wiki/Model-driven_engineering [Accedido febrero 16, 2013].

[2] Wikipedia, QVT1.1. Available at: http://www.omg.org/spec/QVT/ [Accedido febrero 16, 2013].

[3] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev. "ATL: a model transformation tool". 2008. Science of Computer Programming 72:31-39 (Elsevier). See also http://www.eclipse.org/atl/

[4] A. Sch¨urr. Specification of graph translators with triple graph grammars. In *WG'94*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[5] D. S. Kolovos, R. F. Paige, and F. Polack. The Epsilon Transformation Language. In ICMT'08, volume 5063 of LNCS, pages 46–60. Springer, 2008. Available at : http://www.eclipse.org/epsilon/doc/etl/

[6] Esther Guerra, Juan de Lara, Dimitrios S. Kolovos, Richard F. Paige and Osmar Marchi dos Santos. "Engineering model transformations with transML". 2011. Software and Systems Modeling (Springer). In press.

[7] Wikipedia, Eclipse. Available at: http://es.wikipedia.org/wiki/Eclipse_%28software%29 [Accedido febrero 16, 2013].

[8] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks."EMF: Eclipse Modeling Framework", 2nd Edition. Addison-Wesley Professional, 2008.. Available at: http://www.eclipse.org/modeling/emf/ [Accedido febrero 16, 2013].

[9] Graphiti, a Graphical Tooling Infraestructure. Available at: http://www.eclipse.org/graphiti/ [Accedido febrero 16, 2013].

[10] Graphical Modeling Framework. Available at: http://www.eclipse.org/modeling/gmp/ [Accedido febrero 16, 2013].

[11] Dimitrios S. Kolovos, Louis M. Rose, Saad Bin Abid, Richard F. Paige, Fiona A. C. Polack, Goetz Botterweck. "Taming EMF and GMF Using Model Transformation". 2010. LNCS 6394, pp. 211-225. Available at http://www.eclipse.org/epsilon/doc/eugenia/ [Accedido febrero 16, 2013].

[12] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos,Fiona A. C. Polack. "The Epsilon Generation Language". 2008.LNCS 5095, pp.: 1-16. Available at http://www.eclipse.org/epsilon/doc/egl/ [Accedido febrero 16, 2013].

[13] Acceleo. Available at http://www.eclipse.org/acceleo/ [Accedido febrero 16, 2013].

[14] Jet. Available at http://www.eclipse.org/modeling/m2t/?project=jet [Accedido febrero 16, 2013].

[15] Epsilon Book. Available at http://www.eclipse.org/epsilon/doc/book/ [Accedido febrero 16, 2013].

[16] Graphiti developer guide. Available at http://help.eclipse.org/juno/index.jsp?nav=%2F27 [Accedido febrero 16, 2013].